2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN OF A DL/I-TO-NETWORK INTERFACE FOR THE MULTI-MODEL, MULTI-LINGUAL, MULTI-BACKEND DATABASE SYSTEM

by

William A. Sheehan

December 1989

Thesis Advisor David K. Hsiao

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified		1b Restrictive Markings		
2a Security Classification Authority		3 Distribution Availability of Report Approved for public release; distribution is unlimited.		
2b Declassification Downgrading Schedule				
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)		
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol *(if applicable)* 52	7a Name of Monitoring Organization Naval Postgraduate School		
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000		7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000		
8a Name of Funding Sponsoring Organization	8b Office Symbol *(if applicable)*	9 Procurement Instrument Identification Number		
8c Address *(city, state, and ZIP code)*		10 Source of Funding Numbers		
		Program Element No	Project No	Task No / Work Unit Accession No
11 Title *(include security classification)*				
12 Personal Author(s) William A. Sheehan				
13a Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report *(year, month, day)* December 1989	15 Page Count 124	
16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				

17 Cosati Codes			18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	Cross-access, Multi-Model and Multi-Lingual Database System

19 Abstract *(continue on reverse if necessary and identify by block number)*

There has been a tremendous growth in recent years in the use of data base management systems (DBMS) throughout the world. This has lead to efforts to increase the effectiveness and efficiency of systems designed to create and maintain large databases. The traditional approach has been to select a data model and its associated model-based data language and implement a database system based on that single model. The multi-model and multi-lingual database system (MM&MLDS) was designed to increase the functionality of database systems by allowing the use of multiple data models and several model-based languages on a single system. With this approach, the system could support a heterogeneous collection of databases, each based on the model most appropriate for the individual application requirements.

The current implementation of MM&MLDS is restricted in cross-model accessing the available databases. This thesis is part of the effort to remove these restrictions, thereby allowing the databases based on given models to be accessed by database languages associated with different models. The goal of this thesis is to further increase the functionality of MM&MLDS by permitting a user knowledgeable only in a hierarchical-based data language (DL/I) to access and manipulate information in a network database, while strictly maintaining the integrity of the network model. The emphasis is to provide the design analysis necessary to accomplish the translation. More specifically, to develop a process for transforming a network database schema into an equivalent hierarchical schema and to analyze the DL/I requests that are used to access a database and provide a methodology for equivalent access to a network-based database system.

20 Distribution/Availability of Abstract ☒ unclassified/unlimited ☐ same as report ☐ DTIC users	21 Abstract Security Classification Unclassified	
22a Name of Responsible Individual David K. Hsiao	22b Telephone *(include Area code)* (408) 646-2253	22c Office Symbol 52Hq

by

William A. Sheehan
Lieutenant, United States Navy
B.A., Saint Michael's College, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1989

Author: William Sheehan
William A. Sheehan

Approved by: David K. Hsiao
David K. Hsiao, Thesis Advisor

[illegible]
Thomas Wu, Second Reader

Robert B. McGhee
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

There has been a tremendous growth in recent years in the use of data base management systems (DBMS) throughout the world. This has lead to efforts to increase the effectiveness and efficiency of systems designed to create and maintain large databases. The traditional approach has been to select a data model and its associated model-based data language and implement a database system based on that single model. The multi-model and multi-lingual database system (MM&MLDS) was designed to increase the functionality of database systems by allowing the use of multiple data models and several model-based languages on a single system. With this approach, the system could support a heterogeneous collection of databases, each based on the model most appropriate for the individual application requirements.

The current implementation of MM&MLDS is restricted in cross-model accessing the available databases. This thesis is part of the effort to remove these restrictions, thereby allowing the databases based on given models to be accessed by database languages associated with different models. The goal of this thesis is to further increase the functionality of MM&MLDS by permitting a user knowledgeable only in a hierarchical-based data language (DL/I) to access and manipulate information in a network database, while strictly maintaining the integrity of the network model. The emphasis is to provide the design analysis necessary to accomplish the translation. More specifically, to develop a process for transforming a network database schema into an equivalent hierarchical schema and to analyze the DL/I requests that are used to access a database and provide a methodology for equivalent access to a network-based database system.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. MOTIVATION

The traditional approach to designing and implementing a database system involves as the first step analyzing the structure of the database of a given task and choosing an appropriate data model for the database structure. Candidate models include the hierarchical data model, the relational data model, the network data model, or the entity-relationship model to name just a few. The next step in the process is to specify a data language based on the selected model for the specification of the task over the modeled database, e.g., DL/I for the hierarchical data model or SQL for the relational model.

A number of database systems have been developed utilizing this traditional approach. For example, IBM introduced the Information Management System (IMS) in the sixties, which supports the hierarchical data model and the hierarchical-model-based data language, Data Language I (DL/I). Sperry Univac introduced the DMS-1100 in the early seventies, which supports the network data model and the network-model-based data language, CODASYL Data Manipulation Language (CODASYL-DML). And more recently, there has been IBM's introduction of the SQL/Data System which supports the relational data model and the relational-model-based data language, Structured English Query Language (SQL).

Each of these traditional database designs can be characterized as being a *mono-model and mono-lingual database system.* That is, each is based on a single data model that acts as a high-level abstraction of the underlying data that makes up the database itself. The user interacts with the database by writing transactions in the model-specific data language designed to support that data model. The obvious limitation of this approach is that the user is restricted to a single data model and a specific model-based data language.

A much more flexible approach to database design was proposed by Demurjian and Hsiao [1]. Modern database systems should support and execute a large number of shared databases using a finite set of data models and associated data languages. Such a system is called a *multi-model and multi-lingual database system (MM&MLDS).*

There is a number of distinct advantages to such an approach. Perhaps the most practical of these involves the reusability of database transactions developed on an existing database system. In MM&MLDS, there is no need for the user to convert a transaction from one data language to another. MM&MLDS allows the running of database transactions written in different data languages. Therefore, the user does not have to perform either manual or automated translation of existing transactions in order to execute a transaction in MM&MLDS. MM&MLDS provides the same results even if the data language of the transaction originates at a different database system.

Another distinct advantage is in support and training. Since MM&MLDS supports a variety of data models and languages on a single system, existing employee skills can be utilized on the multiple data models reducing overall training costs. Additionally, database resources are specialized to the particular mix of requirements within an organization rather than relying on a single, mono-model, mono-lingual system that must attempt to be general enough to handle diverse requirements. This specialization results in an increase in both performance and functionality.

A more subtle, but significant advantage of using MM&MLDS involves the flexibility to explore the effectiveness of various data models for a given database application. The development of a new application might involve parallel implementation of a small number of databases utilizing different data models and languages that appear appropriate to the envisioned use. Further testing and analysis may indicate that the mix of transactions specific to that application are more efficiently and effectively handled by one pair of the selected models and languages.

B. THE MULTI-MODEL AND MULTI-LINGUAL DATABASE SYSTEM

A block diagram of the structure of a multi-model and multi-lingual database system is shown in Figure 1. A user accesses and modifies the database by interaction with the *language interface layer (LIL)* through a specific *user data model (UDM)*. Transactions are written in a *user data language (UDL)* defined for the chosen model. Transactions are of two general types, database definition requests and database manipulation requests. LIL identifies which of these types is currently being input by the user and routes the transaction sequences to the *kernel mapping system (KMS)* for processing.

KMS handles the requests in two ways. If the transactions are database definition requests, KMS transforms the UDM database definition to a *kernel data model (KDM)* database definition equivalent. The transformed definition is then forwarded to the *kernel controller (KC)* which, in turn, routes the requests to the *kernel database system (KDS)* for processing. When KDS has finished with the database creation, it notifies

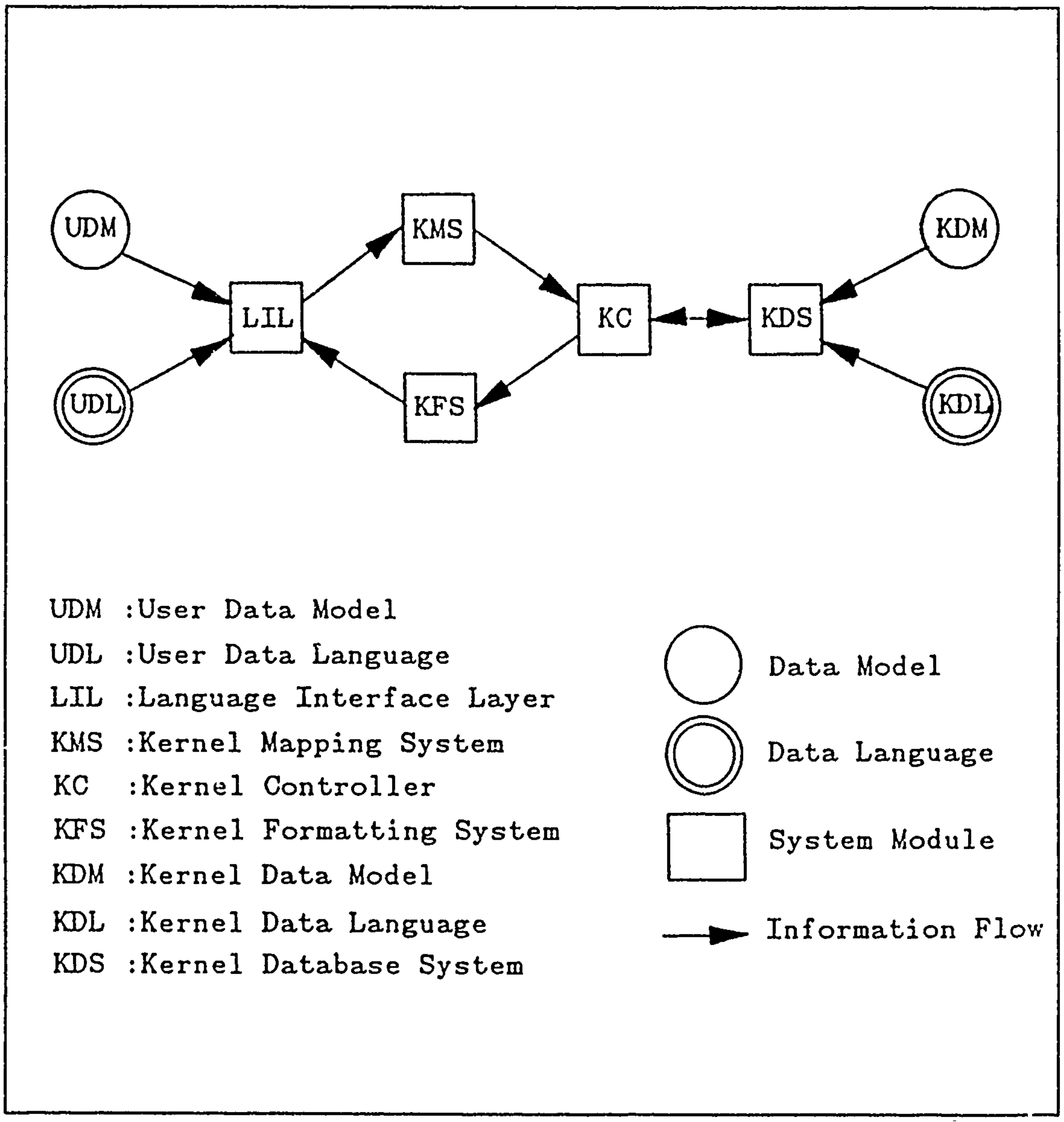


Figure 1. The Multi-Model and Multi-Lingual Database System (MM&MLDS)

KC of the completion, which in turn notifies the user through LIL that the database definition request has been processed and further requests can be accepted.

If the transactions are database manipulation requests, KMS transforms the UDL transactions to their KDL equivalents. These requests are then forwarded to KDS, through KC, for processing. KDS returns the results of the transaction to KC. KC forwards these results to the *kernel formatting system (KFS)* where they are transformed from their KDM structure to a UDM equivalent. The results are then displayed to the user in a format consistent with UDM.

LIL, KMS, KFS, and KC define the language interface for a single user-defined data model and data language. In a multi-model and multi-lingual database system, a separate interface is required for each model and language pair defined, as shown in Figure 2. For example, in the current system, a unique language interface has been developed for the relational/SQL pair, the hierarchical/DL/I pair, the network/CODASYL-DML pair, and the functional/Daplex pair. In contrast, KDS structure is a single, common component shared by all pairs. It is through KDS that the physical database is accessed and manipulated by the various user-defined model/language interfaces.

The attribute-based data model (ABDM) and attribute-based data language (ABDL) have been implemented as KDM and KDL, respectively, for MM&MLDS. ABDM is a simple yet powerful data model first described by Hsiao[2,3] and studied by Rollins[4]. Subsequent reports have been completed which show how relational[5], hierarchical[6], and network[7] constructs can be mapped to attribute-based equivalents. The background information for the data-language interfaces from SQL to ABDL[8,9], from DL/I to ABDL[10,11] and from CODASYL-DML to ABDL[12,13] have been published.

C. THE MULTI-BACKEND DATABASE SYSTEM

The *multi-backend database system (MBDS)* has been designed to overcome the performance and replacement problems associated with a traditional mainframe-based approach to database system design. MBDS has solved these problems by moving the database functions to a separate system with its own dedicated hardware and software. As shown in Figure 3, the MBDS controller is a separate computer from the other computers, known as *backends.* It acts as an interface to a host computer or directly to users and performs the controlling functions of the database system. Transactions are passed to the backends and the results of database operations are routed back to the controller from the backends. The backends are the database engines of MBDS. Each is a separate mini- or micro-computer connected in parallel via a broadcast communications bus. Each backend maintains a portion of the database on one or more hard disk subsystems. This parallelism proves to be the key to the high-performance of the system. When a transaction is broadcast by the controller, each backend can execute the transaction on its portion of the database, independent of the other backends.

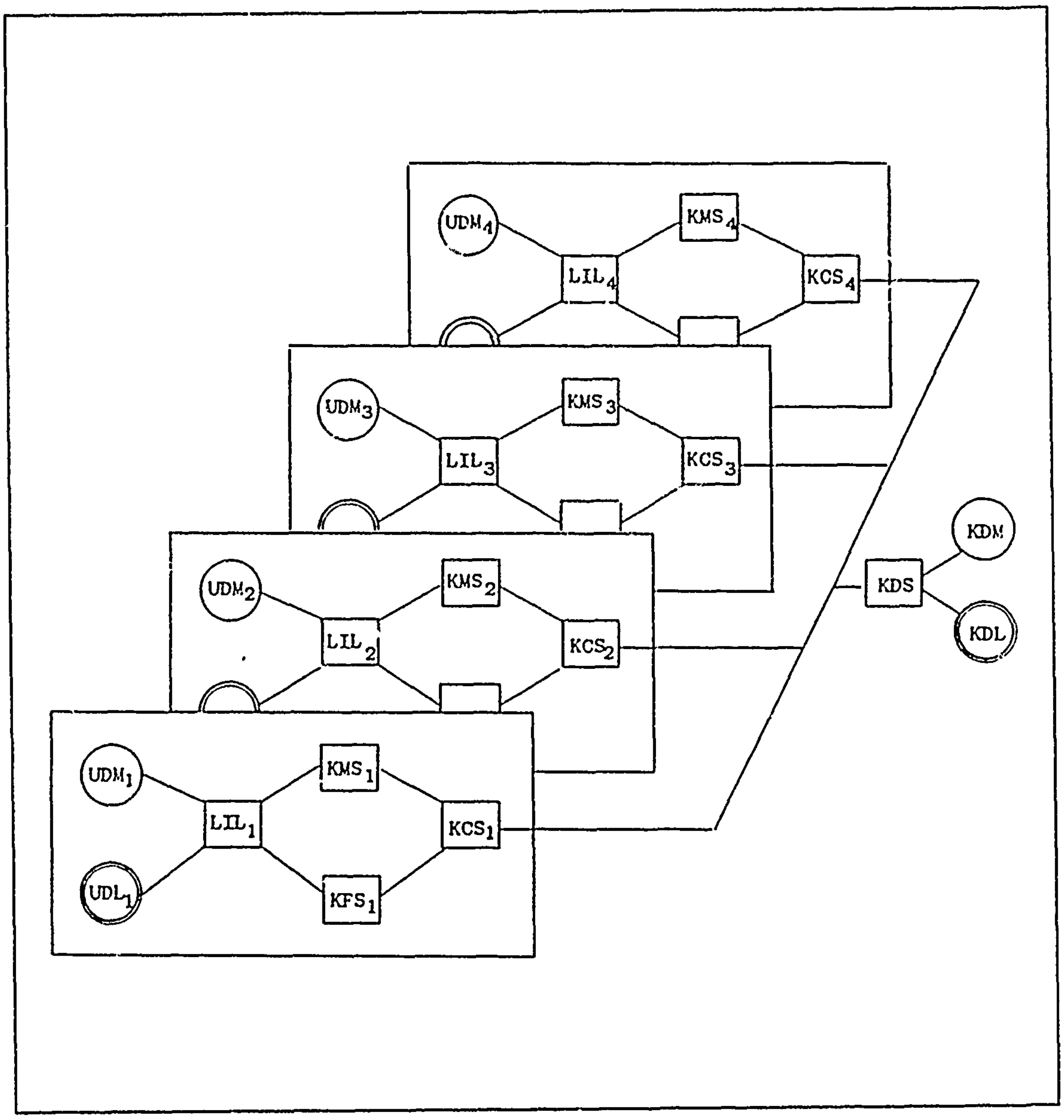


Figure 2. Multiple Model/Language Interfaces

The benefits of the MBDS architecture lie in its capability to provide performance gains and to accommodate database growth. Performance gains can be realized by increasing the number of database backends. Assuming a constant size of the database, MBDS should produce a nearly proportional decrease in response times when the number of backends is increased. Additionally, a proportional increase in the number of

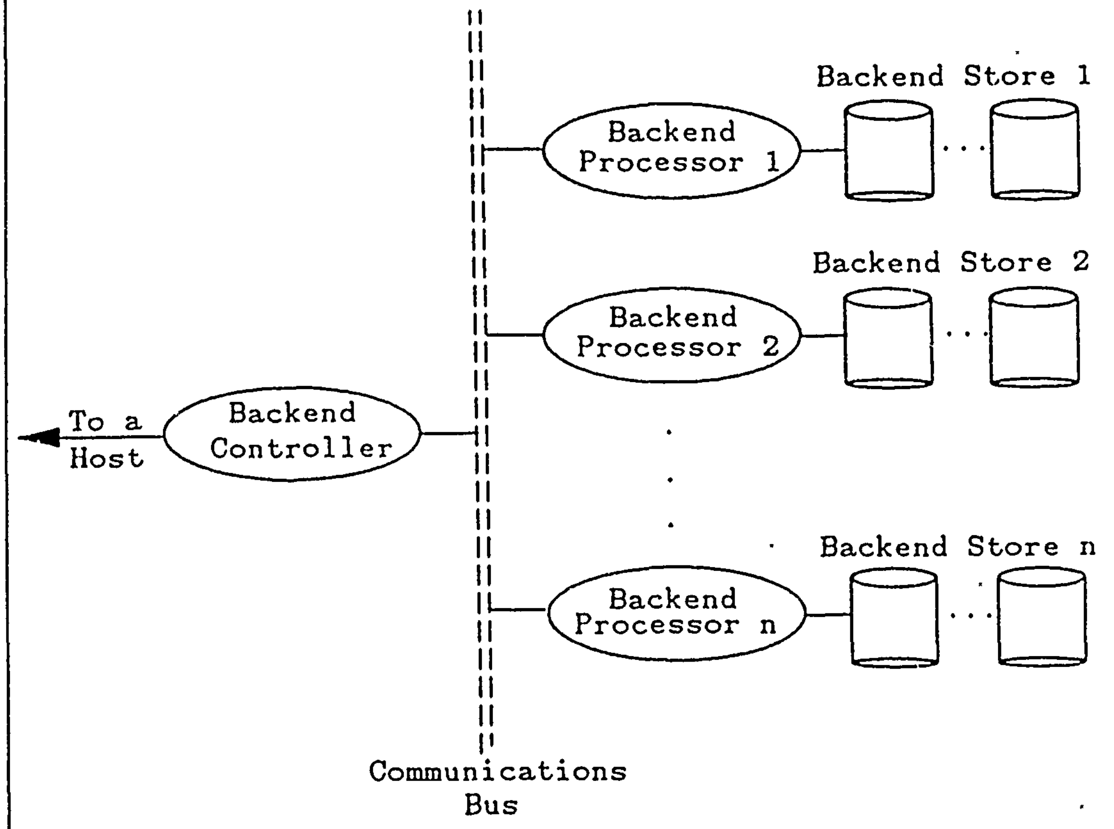


Figure 3. The Multi-Backend Database System

backends in relation to an increase in the size of the database produces nearly invariant response times for a given set of transactions.

MBDS also provides a high degree of extensibility. The system can accomodate additional backends with no modification to existing software and no new programming. In addition, no modification to existing hardware is necessary and the disruption of system activity is minimal.

D. AN OVERVIEW

The current implementation of MM&MLDS is restricted in cross-model accessing the available databases. This thesis is part of the effort to remove these restrictions, thereby allowing the databases based on given models to be accessed by database languages associated with different models. This extends the multi-model and multi-lingual database system into cross-model accessing capabilities.

In particular, we are concerned in this thesis with the design and analysis of a methodology which will allow a DL/I user to access a network database. We note that DL/I is a hierarchical data language traditionally associated with the hierarchical model. Now, we want it to be able to access databases modeled in the network way. In chapter II, we describe the attribute-based, hierarchical, and network models and their associated languages in order to provide a basis of understanding for the subsequent discussions. In Chapter III, we examine a number of strategies for implementing the cross-access of a network database via DL/I transactions in the MM&MLDS and select the most appropriate approach. Chapter IV details the implementation issues involved in transforming a network schema to functionally equivalent hierarchical schemas; and in Chapter V, we discuss the design and implementation issues involved in transforming DL/I transactions into ABDL equivalents that will allow manipulation of data in a network database while maintaining the integrity of that database. Finally, in Chapter VI, we provide our conclusions concerning the proposed design.

II. DATA MODELS

In this chapter, we briefly describe the various data models ıd model-based data languages necessary for a full understanding of the multi-model transformation. In section A, we discuss the attribute-based data model (ABDM) and its associated attribute-based data language (ABDL). Section B outlines the hierarchical data model and its associated DL/I data language. Finally, in section C, the network data model and the CODASYL-DML language are discussed.

A. THE ATTRIBUTE-BASED DATA MODEL, ABDM

As stated in Chapter 1, the attribute-based data model and ABDL have been implemented as the KDM and KDL respectively in the multi-model and multi-lingual database system. This model and its associated language, as originally developed by Hsiao[2,3], is a simple yet powerful construct for creating and manipulating databases.

1. Model Description

A *database* consists of a collection of files. Each *file* contains a group of related records. A *record* is made up of a collection of *attribute-value pairs*. An attribute-value pair is a Cartesian product consisting of an attribute name and an attribute value. For example, < GRADE,A > is an attribute-value pair having GRADE as an attribute name and associated attribute value of 'A'. A record may also contain an optional *record body*, containing textual information related to the record. An example of a record, without a record body, is shown below.

```
( <FILE,Student>,<NAME,Ford>,<SNUM,0284>,<GRADE,B> )
```

The first attribute-value pair in each record identifies the file name. In this case, the file name is 'Student'. There is at most one attribute value pair for each unique attribute defined in the database.

Access to the database is through a query of *keyword predicates.* A predicate is a three-tuple in the form < attribute,operator,value >, such as (SNUM < = 0284). A query on a database then, is a finite number of keyword predicates in disjunctive normal form. For example,

```
(((FILE = Student) and (SNAME = Ford)) or
((FILE = Student) and (SNAME = Jones)))
```

2. The Attribute-Based Data Language, ABDL

Access and manipulation of a database are performed through five primary operations (insert, delete, update, retrieve, and retrieve-common). These operations are formed by utilizing the queries as just described. A brief description of each operation follows.

The INSERT request is used to insert a new record into a specified file of an existing database and takes the form:

```
INSERT Record
```

An example of an INSERT operation which inserts a student record into a file named Student is

```
INSERT (<FILE = Student>,<SNAME = Henry>,<SNUM = 3468>)
```

The DELETE operation is used to remove one or more records from the database. A DELETE operation takes the form:

```
DELETE Query
```

An example of a DELETE which removes all students named 'Hayes' from the Student file is:

```
DELETE ((FILE = Student) and (SNAME = Hayes))
```

An UPDATE is used to modify records of the database. An UPDATE request consists of two parts. The syntax is:

```
UPDATE Query Modifier
```

An example of an UPDATE request which changes the grade of a student named Oliver to an 'A' is:

```
UPDATE ((FILE = Student) and (SNAME = Oliver) (GRADE = A))
```

A RETRIEVE request is used to retrieve records from the database. The database is not altered by this operation. A RETRIEVE consists of three parts, a query, a target-list and an optional by-clause. The target list specifies the set of attributes to be output to the user. It may consist of an aggregate operation (avg, count, sum, min, max). The by-clause is used to group the output records. The syntax for a RETRIEVE request is:

```
RETRIEVE (Query) (Target-list) (by-clause)
```

For example:

```
RETRIEVE (FILE = Student) (SNAME) BY SNUM
```

would retrieve the names of all students, ordered by their student number.

The final operation is the RETRIEVE-COMMON request. It is used to merge two files by common attribute values. The syntax for a RETRIEVE-COMMON request is:

```
RETRIEVE (Query 1) (target-list 1)
COMMON (attribute 1, attribute 2)
RETRIEVE (Query 2) (target-list 2)
```

An example of such a request is:

```
RETRIEVE (FILE = STUDENT) (SNAME)
COMMON (SNAME,TNAME)
RETRIEVE (FILE = TEACHER) (TNAME)
```

This request would display a list of students and teachers that share a common name. As with the retrieve command, the database is not modified by this operation.

B. THE HIERARCHICAL DATA MODEL

1. Model Description

A hierarchical database is composed of an ordered set of trees. A *tree* consists of a single, root record type with an ordered set of one or more dependent subtrees. Each subtree in turn consists of a single root record and set of zero or more dependent subtrees. Hierarchical structures are a very natural way to model real-world systems such as business organizations, military chains of commands, university course offerings, etc. and thus are ideal structures for database organization. Each record type is composed of one or more attributes which uniquely define it. A record type is connected to dependent record types through directed arcs or links. In a hierarchical model, the links define a one-to-many relationship from the parent to the child record type. At most, one link can exist between two record types. One of the key constraints in a hierarchical system is that *no occurrence of a child record type can exist without its parent.* This implies that many of the operations on a database must necessarily affect record occurrences other than those specifically identified. For example, if a record occurrence is deleted from the database, the entire subtree consisting of dependent child records to which it is linked must also be deleted. Similarly, a child record occurrence cannot be inserted into the database unless its parent currently exists.

2. The Data Manipulation Language, DL/I

One of the first, and possibly still the most utilized, database system was introduced in 1968 by International Business Machines (IBM) under the product name Information Management System (IMS)[14]. An IMS database consists of a hierarchical arrangement of *segments* (records), each of which is composed of a collection of *fields.* The data manipulation language utilized by IMS is called Data Language, One (DL/I). Queries to a hierarchical database are designed to be made by issuing DL/I calls from within a host language such as COBOL or PL/I. Since MM&MLDS is a stand-alone system, there is no need for such a host language. Therefore, a more descriptive syntax has been implemented following the general form outlined by Date[15]. Four basic operations (get, insert, delete, and replace) have been implemented within MM&MLDS.

The GET operations (GU, GHU, GNP, GHN, etc.) are used to set currency pointers within a hierarchical database and perform retrieval of segment occurrences.

Various forms of this operation are also utilized to prepare the database for other manipulation commands. The following example is a Get Unique (GU) query which is used to retrieve the first student occurrence with a grade of B in course number C100 taught in July 1989.

```
GU course (cnum = 'c100')
offering (date = '0789')
student (grade = 'B')
```

In addition to the record retrieval, currency pointers have been set within the database and retrieval of additional records meeting the same criteria can be accomplished through looping constructs utilizing a Get Next (GN) operation. For example, the following loop transaction will retrieve the remaining records meeting the above constraints.

```
aa GN student
GOTO aa
```

An INSERT operation is accomplished by specifying a record occurrence and then identifying the hierarchical path to the desired insertion point. For example, the following query inserts a record in the offering segment for course C100, identifying the date, location, and format of that course.

```
BUILD (date,location,format) : ('0789','S123','lecture')
ISRT course (cnum = 'c100')
offering
```

DELETE operations are performed by setting database currency pointers via a Get Hold Unique (GHU) operation and then issuing a delete command. The following query deletes a student named 'Burke' from course C100 taught in July 1989.

```
GHU course (cnum = 'c100')
offering (date = '0789')
student (sname = 'Burke')
DLET
```

As previously mentioned, a DELETE operation automatically deletes all dependent occurrences in the hierarchical database.

The REPLACE operation is used to modify an occurrence within a database and can be accomplished by setting the database currency pointers via a Get Hold Unique (GHU) operation, identifying the field to change, and issuing a replace command. For example, to change the prerequisite for the advanced database course from AI to Data Structures, the following query can be input.

```
GHU course (xtitle = 'Adv. Database')
prereq (ptitle = 'AI')
CHANGE ptitle to 'Data Structures'
REPL
```

A constraint placed on the REPLACE operation in both IMS and MM&MLDS is that a sequence (key) field cannot be updated. A desired change to a sequence field must be accomplished through the use of DELETE and INSERT operations.

3. The Attribute-Based Hierarchical Database, AB(Hierarchical) Database

Of great importance is the mapping of a hierarchical schema to an attribute-based (AB) schema. This mapping should maintain the characteristics and constraints of the hierarchical schema within the AB schema. Since a hierarchical database is stored in the database system as an attribute-based database, the storage structure of the database is characterized by the ABDM wheras the logical organization of the database is characterized by the hierarchical data model. Thus, when we say a hierarchical schema of a hierarchical database in the multi-model and multi-lingual database system, we mean the hierarchical schema of a hierarchical database in the attribute-based form, i.e., AB(hierarchical) schema of a AB(hierarchical) database.

Using a procedure originally outlined by Banerjee [6], we can map the sample hierarchical database of Figure 4 into its ABDL counterpart. However, before doing so we must introduce and explain two notions whose existence are necessary to conduct the data conversion. These notions are that of the IMS *current position* and that of the interface *symbolic identifier.*

IMS uses a pre-ordered traversal to navigate a database tree. Quite understandably, this traversal need not begin at the root each time a call is made to the database. The traversal could easily begin at a child segment. Indeed, the segment requested could be a twin of the segment just previously retrieved. Therefore, it is im-

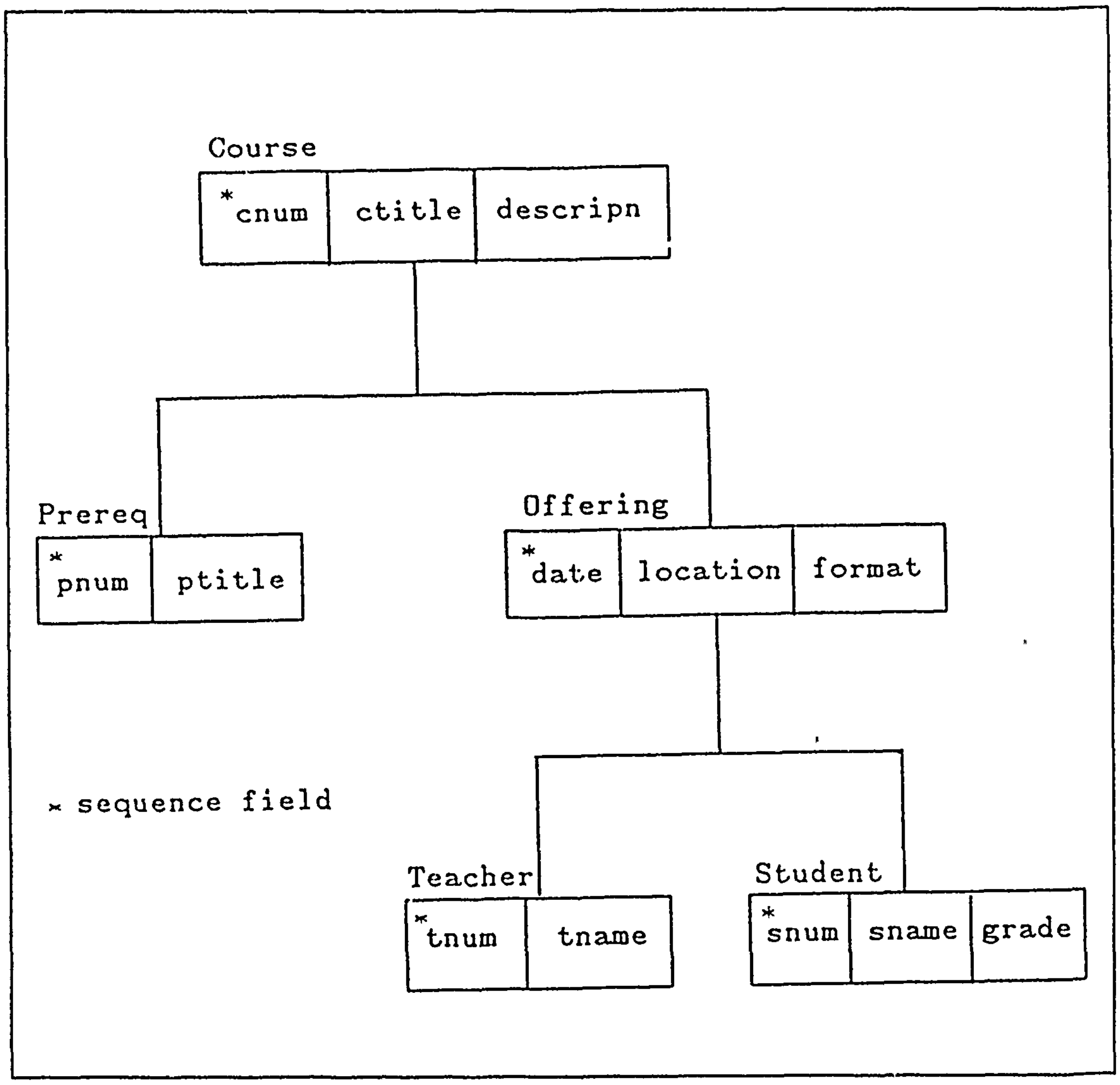


Figure 4. The Logical Data Structure of a Hierarchical Database

portant to know the path of the traversal when conducting DL/I data manipulation operations. This is accomplished by designating the segment upon which the traversal has stopped as the *current position*. The current position of the IMS database is established after each retrieval or insertion operation. For a retrieval operation the current position is the segment just retrieved; for an insertion operation, the current position is the segment just inserted.

In IMS it is necessary to indicate order among twin segments. This is achieved by designating a sequence field in the segment. As we convert our hierarchical data to attribute-based data, we must also be able to distinguish order among twin segments. Thus, in the conversion process we shall assign a *symbolic identifier* to each record. The symbolic identifier of a record R is a group of fields consisting of:

1) the symbolic identifier of the parent of R;
2) the sequence field of R.

With the inclusion of the above notions, the database translation may now occur. An ABDL record may be created from an IMS segment using the following three step process:

Step 1 : For each field in the segment, form a keyword using the field name as the attribute and the field value as the value.

Step 2 : Form a keyword of the form <TYPE, SEGTYPE> where TYPE is a literal and SEGTYPE is the IMS segment type in consideration.

Step 3 : For each sequence field in the symbolic identifier of the segment, form a keyword using the sequence field name as the attribute and the field value as the value.

The final attribute-based representation of the sample academic database is shown in Figure 5.

C. THE NETWORK DATA MODEL

1. Model Description

In general, the network (CODASYL) data model is based on the concept of directed graphs. The nodes of the graphs usually represent entity types which are described by records, while the arcs of the graphs correspond to relationship types that are represented as connections between records. The CODASYL (Conference on Data System Languages) data model is referred to by Tsichritzis and Lochovsky[16] as the most comprehensive specification of a network data model that exists. It is important to note that MM&MLDS uses only a subset of the entire data model as specified by CODASYL. The specific constructs and clauses used in connection with MM&MLDS

```
(<Type,Course>, <cnum,num_of_course>, <ctitle,course_name>,
          <descripn,description>)

(<Type,Prereq>, <cnum,num_of_course>, <pnum,num_of_prereq>,
          <ptitle,course_name>)

(<Type,Offering>, <cnum,num_of_course>, <date,when>,
          <location,where>, <format,form>)

(<Type,Teacher>, <cnum,num_of_course>, <date,when>,
          <tnum,teacher_id_num>, <tname,teacher_name>)

(<Type,Student>, <cnum,num_of_course>, <date,when>,
          <snum,student_id_num>, <sname,student_name>,
          <grade,student_grade>)
```

Figure 5. The Attribute-Based Representation of the Academic Database

are described clearly and succinctly by Wortherly [12]. In an effort to facilitate understandability, our description of the network data moael follows his work.

CODASYL databases are networks of record types and set types, where records and sets are the entities which describe the databases. A record type in a CODASYL database is a collection of hierarchically related data items or field names[7]. These field names are specified in a schema declaration (template) for that record type. A *record* is any occurrence of a record type and has specific values assigned to the data items named in the schema declarations. This implies that a record type is simply a generic name for all of the records that are described by the same template.

Set types in a CODASYL database indicate relationships between record types. They consist of a single record type called the *owner* record type, and one or more record types called the *member* record types. Thus, a set type expresses explicit associations between different record types in the database. This characteristic makes it possible for a designer to model a large variety of real world database management problems involving diverse record types. Of special importance here is the fact that the owner record type of a set type is not allowed to be a member of the same set type.

Set types have occurrences just as record types do. Each occurrence of a set type has one occurrence of the owner record type and zero or more occurrences of each of its member record types. Again it must be noted that a record occurrence cannot be present in two different occurrences of the same set type. This qualification emphasizes the pairwise disjointness of set occurrences of a given set type. Figure 6 gives an example of a set occurrence involving an owner record occurrence and two member record occurrences.

2. The Data Manipulation Language, CODASYL/DML

CODASYL-DML is a procedural data language. The user of a CODASYL database writes his programs in a general purpose language that hosts the CODASYL-DML. In general, most operations in a CODASYL database are carried out by "navigating" through set occurrences. The starting point for this navigation is usually the current record of the run unit. The *run unit* is the application program (transaction) being executed. Other DML operations can be based on the current record occurrence of a set type or record type.

CODASYL-DML has several primary operations which support the primary database operations of retrieval, insertion, deletion, and modification (updating existing records). Different implementations provide varying collections of these operations, but we will concentrate our discussion on the basic ones.

The cornerstone of CODASYL-DML is the FIND statement. This statement is used to establish the currency of the run unit, and optionally used to establish the currency of the set type and the record type. The general format of the FIND statement is

```
FIND record-selection-expression [ ] ,
```

where the square brackets contain optional expressions for the suppression of updates to the currency indicators. In other words, we may suppress the updating of the currency for a record type, a set type, or both. The record-selection-expression has several different forms each designed to access a particular record in three different ways:

```
        S (an owner record occurrence)
              +-------------------------+
              | S2 | Jones | 10 | Paris |
              +-------------------------+

                  (a set occurrence)
                       (S-SP)

              (two member record occurrences)

                 SP                SP
      +---------------+    +---------------+
      | S2 | P1 | 300 |    | S2 | P2 | 400 |
      +---------------+    +---------------+
```

Figure 6. A CODASYL Set Occurrence

without reference to a previously accessed record; relative to a previously accessed record; or by repetition.

The GET statement in CODASYL-DML complements the FIND statement. Once a record is found, the GET statement places the record in the transaction's working area for access by the transaction. There are two basic formats for the GET statement. They include GET record_type, which gives the transaction access to the entire record, and GET items IN record_type, which gives access to only requested data items in the record type.

The STORE statement is used to place a new record occurrence into the database. The programmer must build up an image of the record prior to the STORE request using assignment statements which are a part of the host language in which the CODASYL-DML is embedded. Once the record image has been created, the proper record must be selected by the database management system.

The set occurrence in which the new record is stored is determined by the SET SELECTION clause specified in the schema definition for the object database. The three options available are: BY APPLICATION, which means that the application program (transaction) is responsible for selecting the correct occurrence; BY VALUE, which means the system selects the proper occurrence based on data item values specified to the owner of the set occurrence desired; and, BY STRUCTURAL, which means that the system selects an occurrence by locating the owner record with a specific item value equal to the value of that same item in the record being stored. The restriction on the last two options is that the data items being used must have been specified with DUPLICATES NOT ALLOWED in the schema definition.

If the user transaction desires to manually insert records into the database, two requirements exist. First, the schema definition must include the INSERTION IS MANUAL clause in the set description for this particular member record. Then the CONNECT statement is used, instead of the STORE statement, for insertion of the record into the database. The record to be inserted is the current record of the run unit. The set occurrence in which the record is inserted is determined in the same way as the STORE statement.

There is also a statement in the CODASYL-DML which performs the opposite operation, namely, the manual removal of a record occurrence from a set. The DISCONNECT statement performs this operation. It disconnects the current record of the run unit from the occurrence of the specified set that contains the record. The record occurrence still resides in the database, but it is no longer a member of the specified set. There is a qualification involved with this statement, however. The record to be disconnected must have a RETENTION IS OPTIONAL clause in the member description for the set type in the schema.

In order to delete records from a CODASYL database, the ERASE statement is used. There are four basic options to this statement; but two are not used in connection with MM&MLDS. The simplest of the two used in MM&MLDS is the ERASE without the ALL option. This statement causes the current record of the run unit to be deleted from the database if, and only if, it is *not* the owner of a non-empty set. If it is the owner of a non-empty set, the ERASE fails.

The ERASE ALL option causes the current record of the run unit to be deleted whether or not it is the owner of a non-empty set. Additionally, this option causes each member record of the set to be deleted, and if they too are owners of non-empty sets, their members are deleted. This action continues all the way down the hierarchy. As one can see, an entire database could be destroyed if the user is not diligent when using this option.

The final statement included in the MM&MLDS subset of CODASYL is the MODIFY statement. It is used to modify values of data items in a record occurrence. This includes modifying all data items or any subset of the data items in the record type. It may also be used to change the membership of a record occurrence from one set occurrence to another, as long as, they are of the same set type. Thus, we have our basic working set of CODASYL-DML statements.

3. The Attribute-Based Network Database, AB(Network) Database

Of great importance is the mapping of a network schema to an attribute-based (AB) schema. This mapping should maintain the characteristics and constraints of the network schema within the AB schema.

Using a modification of a procedure originally outlined by Banerjee [7], the transformation of network data into attribute-based data becomes a relatively straightforward task. The data must be transformed into records which consist of a set of variable-length attribute-value pairs and a record body. The attribute-value pairs may represent the type, quantity, or characteristic of the value, and the record body is as described in section A. Additionally, all attributes in the attribute-based records are distinct, for logical reasons.

The key aspect of the mapping process is the retention of the CODASYL notion of records and sets (the linkages among records). We emphasize that the CODASYL notion of records and sets are *not* the same as the attribute-based notion of records and sets. Therefore, the mapping algorithm presented here uses the attribute-based constructs (or notions) to implement the CODASYL notions.

A CODASYL record type is structured as a hierarchical configuration of data items such as depicted in Figure 7(a), where R1 is the record name, and A, B, C, D, E, represent data item names. Figure 7(b) shows an occurrence of record R1. Notice that only the values of the data items are present in the CODASYL record. In the attribute-based system, both the data-item-name and its value are stored in the record.

```
Record R1                     Record R1

                        +------------------+
 01 A                   |    a01_value     |
 01 B                   |    b01_value     |
    02 C                |    c02_value     |
    02 D                |    d02_value     |
       03 E             |    e03_value     |
    02 A                |    a02_value     |
 01 F                   |    f01_value     |
                        +------------------+

 ( a )                        ( b )
```

Figure 7. Hierarchical Structure of a CODASYL record

Thus, in order to capture the CODASYL information, keywords must be created for each of the elementary data items included in the CODASYL record. These data-item keywords should be of the form:

```
< data_item_name, data_item_value >
```

where the data-item-name is qualified by data-item-names at a higher level if it is not unique. Figure 8 shows the data item representation for the CODASYL record of Figure 7. The dots at the beginning of the record and the dots at the end of the record indicate that there are additional keywords generated for the record in order to preserve the CODASYL record information. These additional keywords are explained as follows.

Each record occurrence in a CODASYL database must also belong to a particular type. This implies that a keyword indicating record type must also be included in the attribute-based record. Its format is

```
< TYPE, record_type >
```

where TYPE is a literal.

Finally, each record occurrence of a CODASYL database has a database key (or address) generated for it. Thus, there is a requirement for representation of this value as well in the attribute-based record. The following form is used for this keyword, where DBKEY is a literal.

```
< DBKEY, database_key >
```

So, in representing record information, we have the need for three mandatory keyword types, namely, data-item-name, with or without qualification, TYPE, and DBKEY.

In order for the attribute-based records to be complete, it must also include information related to CODASYL set membeiship, and set ordering. Since occurrences of set types are pairwise disjoint, then each member record occurrence is also identified by its owner record occurrence. This means that we can express set membership by inclusion of the keyword

```
< MEMBER.set_type, owner_database_key >
```

for each set occurrence in which the record is a member.

Finally, the logical position of a record occurrence within a set occurrence is often useful. Thus, ordering of member record occurrences is expressed by inclusion of the keyword

(. . ., < A, a_value >, < B, b_value >,

< C, c_value >, < D, d_value >,

< E, e_value >, < B.A, b.a_value >,

< F, f_value >, . . .)

Figure 8. Attribute-Based Representation of CODASYL Data Items

```
< POSITION. set_type, sequence_number >
```

in the attribute-based record for each set in which the record is a member record.

Therefore, in representing set information, we have the need for two keyword types, those representing member records, and those representing member-record positions within sets.

III. MAPPING FROM THE NETWORK TO THE HIERARCHICAL MODEL

A. MAPPING METHODOLOGIES

As mentioned previously, MM&MLDS is a single database system designed to support a number of different database models and their corresponding data languages. However, MM&MLDS restricts a user to accessing a specific database through the data language implemented to support it. By allowing the databases based on given models to be accessed by database languages associated with different models, we extend the multi-model and multi-lingual database system into cross-model accessing capabilities. For example, a relational user can access a hierarchical database via SQL transactions or a hierarchical user can access a network database using DL/I transactions.

Chapter I outlined the composition of the language interface needed to support each database model. Each interface is specific to the model it supports in terms of capturing the semantics of the data model. Specifically, the attribute-based database created in the data model transformation has the semantics of the corresponding user data model encoded within it. As a result, a given language interface can only access its associated attribute-based database. Notationally, the hierarchical language interface can only access an AB(hierarchical) database and the network language interface can only access an AB(network) database.

In view of these restrictions, we can see that the major challenge for MM&MLDS is to develop a methodology that allows users of one data model to access databases created via the language interface of a different model. More specifically, this thesis focuses on access to a network database by hierarchical users via DL/I transactions.

A number of different design strategies exist for implementing MM&MLDS which can be characterized by the level at which the strategy is integrated into the already existing database system[16]. The basic strategies were first described by Rodeck[17] in terms of accessing functional databases using the CODASYL data language; and further studied by Zawis[18] in terms of accessing hierarchical databases via SQL transactions. The remainder of this chapter summarizes these design strategies and concludes with the selection of the strategy best suited for accessing network databases via DL/I transactions.

1. The High-Level Preprocessing Strategy

The preprocessing strategy is considered to be a high-level process because it occurs on top of the language interface modules as shown in Figure 9. Modifications to the language interface involves three components, a schema transformer, a language translator, and a results formatter. When a user selects a database which is not part of the local language interface (LI), all other LI's are searched in an attempt to find a database. If successful, the schema transformer uses the original database schema to create a parallel and equivalent schema in the local LI, based on the local database model. When the user executes a transaction against this transformed database schema using the local data manipulation language, the language translator generates transactions in the original data manipulation language that can be used to access the database. The results formatter formats the returned responses, if necessary, in the basic form of the local LI.

2. The Mixed-Processing Strategy

The mixed-processing strategy is a mid-level, direct method for the cross-accessing of databases as shown in Figure 10. Two components are involved, a schema transformer and a second language interface. As in the preprocessing strategy, when a user selects a database that is not in the local LI, all other LI's are searched for the desired database. When found, the original database schema is copied and transformed into an equivalent schema in the local LI. When a user executes a transaction in the local data manipulation language, the new language interface processes the request. The AB requests output from this language interface are in the form of the original database model which thereby eliminates the need for an extra language translation step.

3. The Postprocessing Strategy

As shown in Figure 11, the postprocessing strategy is a low-level method for cross-accessing databases. Similar to the pre-processing strategy, three components are involved, a schema transformer, a language translator, and a results formatter. This method is considered low-level because it occurs below the LI's in the kernel database system. In this strategy, the schema transformation occurs from the kernel database schema of the original database to the kernel database schema of the local LI. Language translation occurs in the opposite direction from the kernel database language translations of the local LI to the kernel database language translations of the original database. The results reformatter would then translate the results into the format of the local LI's form.

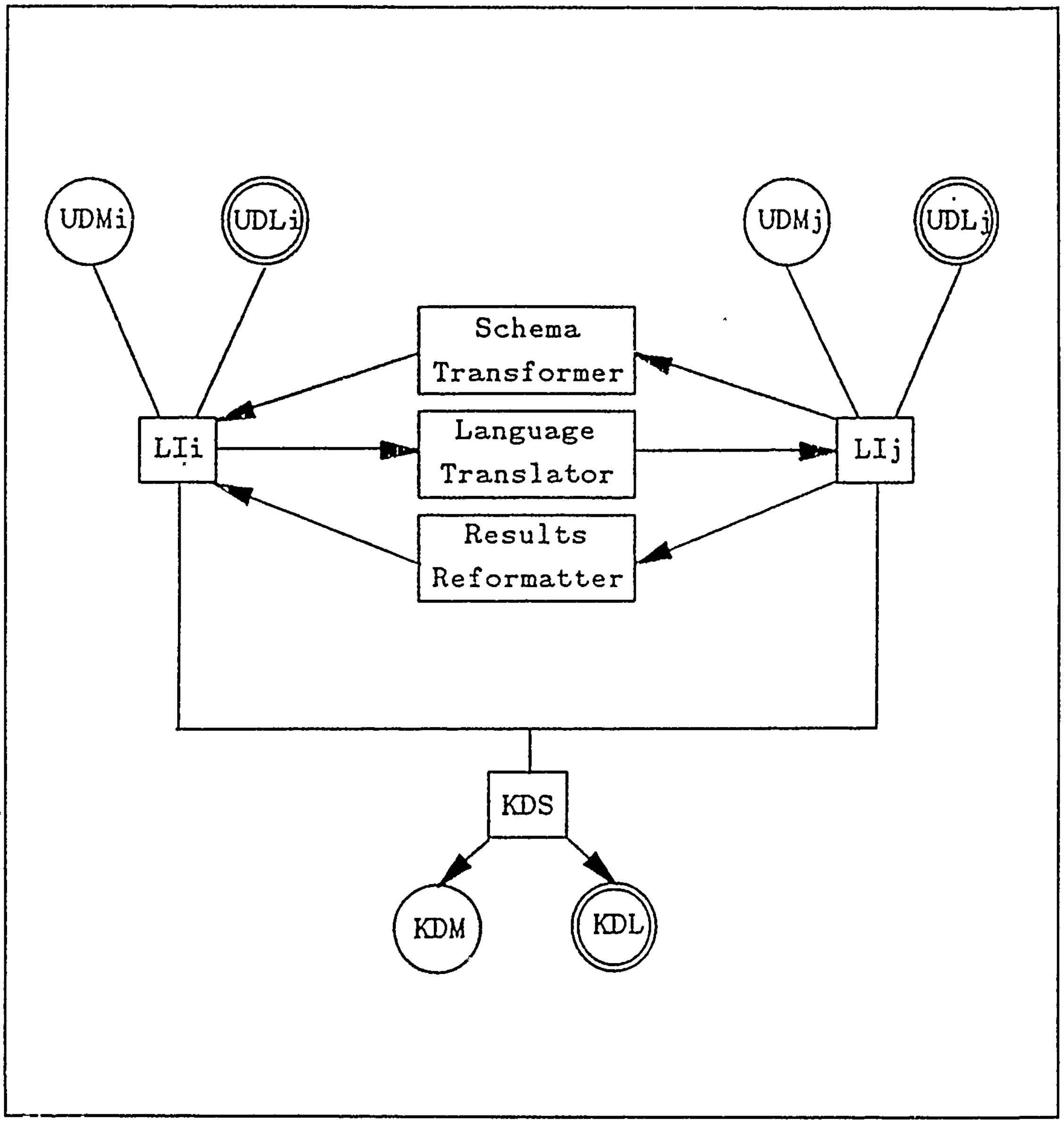


Figure 9. The High-Level Preprocessing Strategy

B. DESIGN CHOICE

In selecting the most appropriate design strategy for accessing a network database via DL/I transactions, we must weigh the advantages and disadvantages of each of the strategies developed. A major problem with the postprocessing strategy is in the location of modifications. This approach deals with the kernel database system and as such, we can expect the focus of programming activity to be in this area. In the current

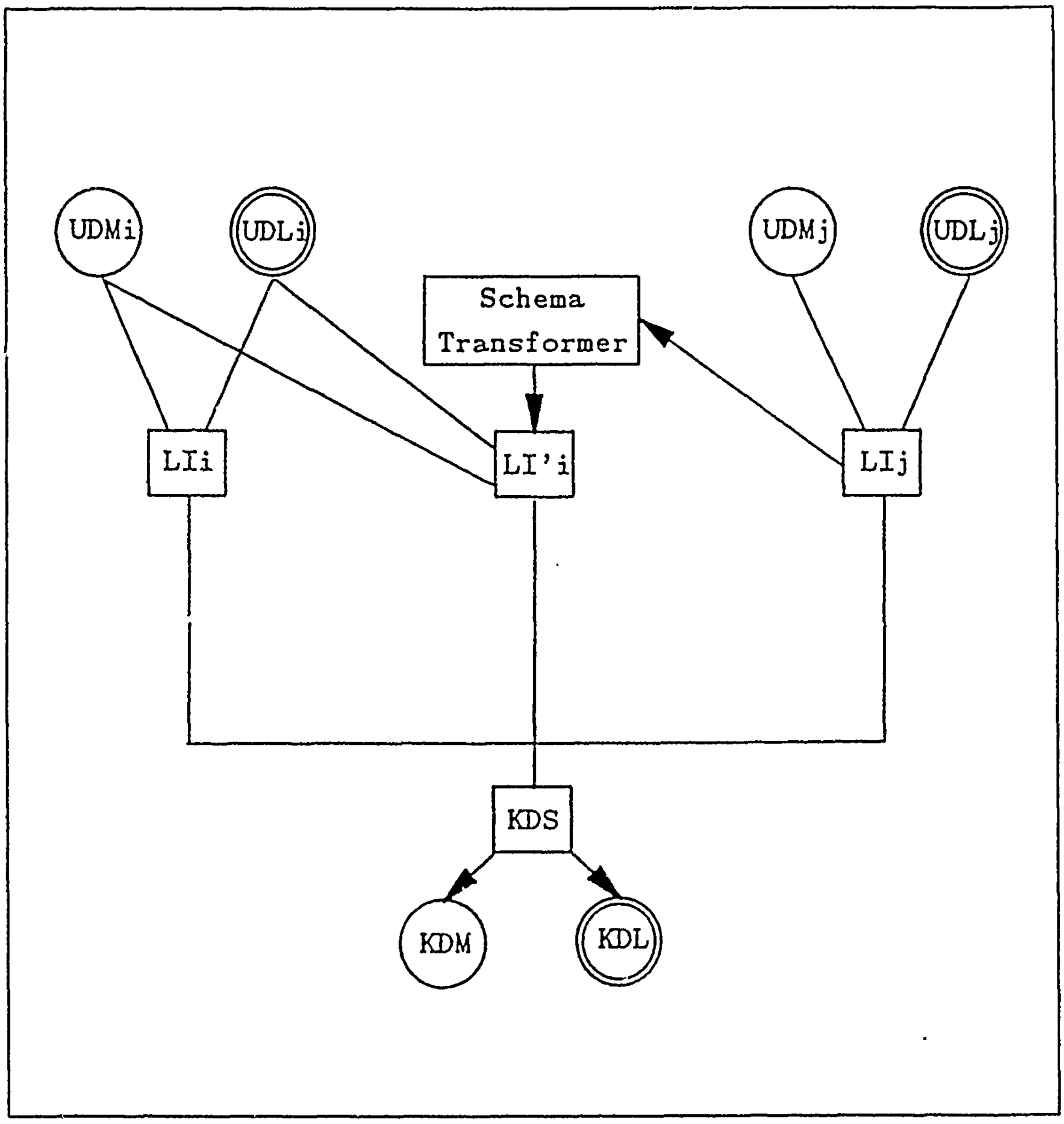


Figure 10. The Mixed-Processing Strategy

implementation of MM&MLDS the kernel database schemas are not visible to the individual language interfaces and implementation of this strategy would force a major design change in the interaction between the kernel database system and the language interfaces. Additionally, the kernel database system was designed as an independent, stand-alone system upon which the MM&MLDS language interfaces were added as a functional enhancement. We find it inadvisable, at this point, to attempt to combine the

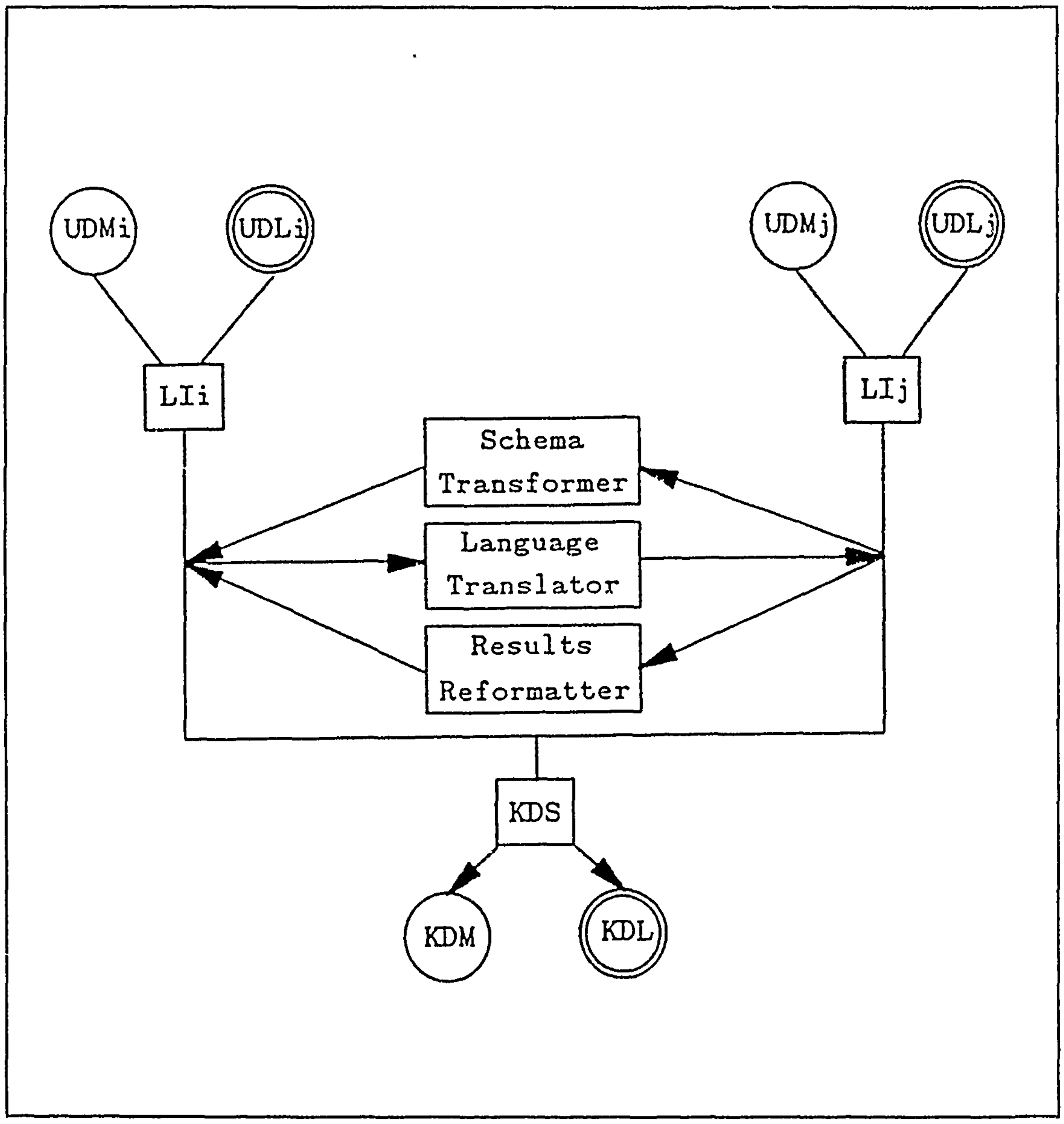


Figure 11. The Postprocessing Strategy

code of these two large projects by coding in an interdependency between specific language interfaces and the kernel database system.

The remaining two strategies, on the other hand, can be implemented completely at the language interface layer of the MM&MLDS. The preprocessing method appears, at least initially, to be conceptually easier to understand and implement by simply converting the transactions input in one data manipulation language into equivalent transactions of a second data manipulation language for access to a database. Upon investigation however, it becomes clear that the task of translating the syntax of a data manipulation language into the syntax of a second language while maintaining the semantic meaning is far from simple.

Additionally, we can expect the overall performance of the preprocessing method to be less than that of the mixed-processing method. In the preprocessing strategy, cross-access of databases requires a schema transformation, two language translations and two reformatting of results. On the other hand, cross-access via the mixed-processing strategy requires one schema translation, one language translation and one reformatting of results. It is clear that less processing activity is needed by the mixed-processing approach resulting in increased performance.

One final point in favor of the mixed-processing method deals with the amount of new code needed and modification to existing code. As outlined earlier in this chapter, the preprocessing strategy requires three components to be implemented, a schema transformer, language translator, and results formatter. These are new software modules to be added to the language interface level. Modifications to the current language interfaces would be relatively minor. The mixed-processing strategy, on the other hand, would require extensive modification to the existing language interface to handle the cross accessing of databases, however, this code would be very similar to the code in the current LI making the implementation task much simpler. We would expect that the amount of code required to implement the mixed-processing strategy to be between one-half to two-thirds of that required to implement the preprocessing strategy.

IV. TRANSFORMING THE AB (NETWORK) SCHEMA TO THE AB (HIERARCHICAL) SCHEMA

A. DESIGN CONSIDERATIONS

Having selected the mixed-processing strategy as the most appropriate for the MM&MLDS design, the first step in implementing this approach, as outlined in the previous chapter, is to perform a schema transformation from a network database to a hierarchical database. This process involves the translation of the relationships implicit in the network database to their functional equivalents in the hierarchical model.

In order to describe this transformation, a sample network database will be used to illustrate the process. Figure 12 shows the schema definition of the Suppliers_and_Parts database. The first line identifies the database name as 'Suppliers_and_Parts'. There are three record types defined in this database (S, P, and SP). These record types will be functionally represented in a hierarchical schema by segment types. 'Duplicates-are-not-Allowed' declarations for SNO and PNO in S and P record types implies these attributes are key fields that uniquely describe an entity or record in question. Therefore, when desiring to insert, the program must check to see if the insert request has an attribute value that already exists in the database. The record attributes are defined by type and length.

A set type is defined by the simple expedience of stating the name of another record type in a record type declaration. DL/I uses the term *parent* rather than the CODASYL term *owner* and the term *child* rather than *member*. In a hierarchical structure, it should be clear that any member record type can have at most one owner.

The set types are now defined. Their purpose is to describe a relationship among record types. The two set types defined are S_SP and P_SP. Each set-type declaration will include the following: owner-record-type name, member-record-type name and insertion and retention rules. The particular details of each set type will differ, depending on the circumstances. The owner-name and member-name statements simply define a static relationship among existing records (i.e., occurrences) of the two record types.

The statement, Insertion is Automatic, in set types S_SP and P_SP, means every record added or modified which represents a record type or subtype, must belong to a particular set. The statement, Retention is Fixed, requires a member record reflecting that a record subtype always belongs to the same owner-record type.

```
Schema Name is Suppliers_and_Parts;
 Record Name is S;
  Duplicates are not Allowed for SNO;
    SNO    ; Type is Character 10.
    SNAME  ; Type is Character 10.

 Record Name is P;
  Duplicates are not Allowed for PNO;
    PNO    ; Type is Character 10.
    PNAME  ; Type is Character 10.

 Record Name is SP;
    QTY    ; Type is Fixed 5.

 Set Name is S_SP;
  Owner is S;
  Member is SP;
    Insertion is Automatic
    Retention is Fixed;
    Set Selection is by Value of SNO in S.

 Set Name is P_SP;
  Owner is P;
  Member is SP;
    Insertion is Automatic
    Retention is Fixed;
    Set Selectio.  s by Value of PNO in P.
```

Figure 12. Suppliers_and_Parts Database Definition

The last statement, Set Selection is by Value, declares that when a record is inserted into a set, the set must be the current set type of SNO in S and, likewise, the current set type of PNO in P. In simpler terms, this means each Supplier and Part will be inserted in the sets based on the owner record types.

In transforming an existing network schema into a functionally equivalent hierarchical schema, various key issues must be observed. It is imperative that certain key constructs are adhered to in order to preserve the structural integrity of the mapped (e.g., network) schema to an equivalent (hierarchical) schema or schemas.

The key issue in the schema transformation from a network database to a hierarchical database is the representation of the owner-set-member relationships in the network database. The many-to-many relationships possible in the network (e.g., a record can have two owners and these owners can have owners) database must be broken into the one-to-many relationships of the hierarchical structure.

In our design, a network owner record type becomes a root segment type in the hierarchical schema. The owner-member relationship is then represented in the hierarchical structure as a parent-child relationship. Each owner record type becomes the root of a hierarchical schema. By cascading data in key fields in the network records to form sequence fields of equivalent hierarchical segments we transform from network to hierarchical. Key fields are defined as fields that uniquely identify the corresponding records. They must remain consistent throughout the transformation.

The obvious disadvantage of this technique is the additional space requirement necessary for the duplication of member record types for different owner record types. The primary advantages, and the reason that this method has been selected for our design, are that this method mirrors the transformation made in the AB(network) schema, and it maintains the integrity of the network structure. Figure 13 details textually, the structure of the transformed schema. It should be noted that the cascaded data in key fields are represented as sequence fields in each of the hierarchical segments, indicating that a value must be specified for these attributes. This becomes essential in maintaining the integrity of the network database when data manipulation is performed using DL/I transactions.

```
dbd name = Suppliers_and_Parts
database type = NETWORK

segm name = S
field name = (SNO,seq), bytes = 4
field name = SNAME,bytes = 10

segm name = S_SP, parent = S
field name = (SNO,seq), bytes = 4
field name = QTY, bytes = 4

segm name = P
field name = (PNO,seq), bytes = 4
field name = PNAME, bytes = 10

segm name = P_SP, parent = P
field name = (PNO,seq), bytes = 4
field name = QTY, bytes = 4
```

Figure 13. Textual Representation of the Hierarchical Schema

B. THE SCHEMA TRANSFORMATION

When a hierarchical user logs onto MM&MLDS, a number of existing data structures are present that contain information relevant to that, and all other, users. The first of these is the dbid_node depicted in Figure 14. This structure points to the linked list of database schemas that have previously been defined in each of the language interfaces. It is through this data structure that a user has access to all of the database currently within the system.

```
union dbid_node
{
 struct  rel_dbid_node  *rel;
 struct  hie_dbid_node  *hie;
 struct  net_dbid_node  *net;
 struct  ent_dbid_node  *ent;
}
```

Figure 14. The dbid_node Data Structure

The hie_dbid_node pointer identifies the first hierarchical database schema. The central data structure for each schema is the hie_dbid_node as shown in Figure 15. This structure contains the database name, number of segments, pointers to the root and current segments, and a pointer to the next database schema. The hie_dbid_node structure has been modified to contain an additional field called dbtype that is used to identify the original database model in which the schema was created. For example, a schema transformation from a network model would include a NET identifier in this field. Additional data structures pointed to by the hie_dbid_node structure completely specify the database schema.

```
struct hie_dbid_node
 {
  char                    name[DBNLength + 1];
  int                     num_seg;
  struct hrec_node       *root_seg;
  struct hrec_node       *curr_seg;
  struct hie_dbid_node   *next_db;
  int dbtype
 }
```

Figure 15. The hie_dbid_node Data Structure

A number of data structures are also created that are specific to the new hierarchical user. The first of these is the user_info data structure shown in Figure 16. This structure uniquely identifies the new user in a multi-user environment and points to the data structures created for the exclusive use of that user. A pointer links this information to the list of data structures associated with all other system users.

```
struct user_info
 {
  char                    uid[UIDLength + 1];
  union   li_info         li_type;
  struct  user_info      *next_user;
 }
```

Figure 16. The user_info Data Structure

Figure 17 depicts the dli_info structure. This is the central data structure created for a hierarchical user and contains much of the information or pointers to information used throughout the user session.

```
struct dli_info
 {
  struct curr_db_info          curr_db;
  struct file_info             file;
  struct tran_info             dli_tran;
  int                          operation;
  struct ddl_info              *ddl_files;
  union  kms_info              kms_data;
  union  kfs_info              kfs_data;
  int                          error;
  int                          answer;
  struct hrec_node             saved_seg_ptr;
  struct hrec_node             saved_seg_ptr2;
  struct Sit_info              *kms_sit;
  struct Sit_info              *sit_iist;
  struct Sit_status_info       *fst_sit_pos;
  struct Sit_status_info       *curr_sit_pos;
  int                          buff_count;
 }
```

Figure 17. The dli_info Data Structure

The Language Interface Layer (LIL) is the primary control module from which all other modules are called. It has been designed to be menu-driven by inputs from the current user. It is through LIL that a user can load new databases, select previously created databases for processing, and access databases by generating and selecting DL/I transactions. Control always returns to LIL following any of these operations. The user may end the current session and return to the operating system by making an appropriate choice from the top level menu.

As previously mentioned, when a new user logs into the system, a number of user-specific data structures are created and initialized. These structures provide the temporary storage necessary for performing various database operations and holding returned results. The first menu presented to the hierarchical user pertains to database selection:

```
Fnter type of operation desired
  (l) - load a new database
  (p) - process old database
  (x) - return to the operating system

ACTION ----> _
```

At this point, the user may choose to load a new database schema, in which case he is prompted to enter the database name and set of creates, or process an already existing schema. If the user chooses to process an existing database schema, he is prompted for the database name. The program will attempt to locate the desired database schema by traversing the linked list of hierarchical hie_dbid_node data structures. If found, the schema is loaded, and query processing may begin.

If the database name is not found, the program searches all other language inte[illegible]r-faces for a matching database name and, if found, copies and transforms the [illegible] schema to a functional equivalent that can be used for access to the associate[illegible] via DL/I transactions. If the user has selected a network database for processing, a new hierarchical schema is created based on the desired network schema. The hierarchical data structures previously discussed have functional equivalents in all other language interfaces. It is through these structures that the transformation is accomplished.

Initially, a new hie_dbid_node is created and attached to the end of the linked list of existing hierarchical schemas. The network database name is then inserted and the schema is tagged as a network equivalent by setting the dbtype variable to 'NET'. A new data structure, hrec_node, shown in Figure 18, is created and attached to the schema. This structure describes each of the segments in a database and is initialized with information available from the equivalent record data structures in the network schema.

```
struct hrec_node
 {
  char                         name[RNLength + 1];
  int                          num_attr;
  struct  hattr_node           *first_attr;
  struct  hattr_node           *curr_attr;
  struct  hrec_node            *parent;
  struct  hrec_node            *first_child;
  struct  hrec_node            *next_sib;
  int                          num_child;
  int                          num_sib;
 }
```

Figure 18. The hrec_node Data Structure

The segment name of the DL/I transaction is compared to the record name in a set type. If a match occurs, then the segment name in the hierarchical schema is equivalent to an owner-record name declared in a set type. If the segment is not found in the set type, then a comparison would be made on the member record. When found, the network record is set equal to the hierarchical record and pointers are set to the first attribute of the hierarchical record and the next sibling.

Each attribute in a segment is fully described by a hattr_node data structure as depicted in Figure 19. Initially, an attribute node is created for each field in a network record. The attribute name, type, and length are transferred directly from the network attribute node. If the network attribute is a key field, then the attribute is flagged in the hierarchical schema with a key attribute having a value of 'true'.

```
struct hattr_node
 {
  char                        name[ANLength + 1];
  char                        type;
  int                         length;
  int                         key_flag;
  int                         multiple;
  struct  hattr_node          *next_attr;
 }
```

Figure 19. The hattr_node Data Structure

The cascading technique is used to map the key from the root nodes to the leaf nodes in the network structure. This is necessary in order to maintain the integrity of the network database. Upon completion of the cascading sequence, the number of attributes is set equal to the number of attributes in the associated network database plus the cascaded fields. Processing continues with subsequent records until the schema is completed. Control is then returned to the LIL for further access and manipulation of the database.

V. MAPPING DL/I STATEMENTS TO AN AB (NETWORK) DATABASE

The previous chapter detailed the schema transformation process necessary for the mixed-processing strategy. The remaining major component required for the cross-access of a network database by way of DL/I transactions is the new language interface (LI). One of the primary purposes of this component is to map the DL/I requests input by the hierarchical user to equivalent AB(network) transactions. This chapter describes the design and analysis of this component. Most of the work in this chapter has already been completed in previous thesis work [11,12, 18], on the MM&MLDS. The particular similarities between CODASYL statements and DL/I statements has been added, along with specific modifications for the network-to-hierarchical translation.

A. THE LI TRANSLATION PROCESS

As stated previously, each database schema created within a given model is transformed intc an equivalent schema in the kernel attribute-based model. This AB schema has unique, embedded structures that ensure that the attribute-based database is the functional equivalent of the database defined in the user defined model. For this reason, the LI must providc a language translation from transactions in the user data manipulation language to the attribute-based transactions specific to that language, e.g., from DL/I to AB(hierarchical). To provide a hierarchical-model user accesses to a network database, it is therefore necessary to create a new language interface that will translate DL/I transactions to their AB(network) equivalents.

This new language interface can be implemented in one of two ways. The first method is to create an entirely separate language interface (LIL, KMS, KC, and KFS), and branch to the appropriate version based on database selection by the user. The alternate approach is to modify the current language interface in such a way that program execution will branch to the appropriate translation and processing activities based on user input. In this manner, a new language interface can be *logically* created without duplication of a large amount of similar code. This reduction in code size is the primary reason why it is recommended to modify existing code in the implementation of the cross-access capability.

1. Query Processing in the LIL

After the user has loaded a new AB(network) database or selected an existing AB(network) database for processing, he is prompted for the mode of query input as follows:

```
Enter mode of input desired
  (f) - read in a group of transactions from a file
  (t) - read in transactions from the terminal
  (x) - return to the previous menu

ACTION ----> _
```

The user now has the option of reading in a group of queries from a prepared file or directly entering the queries from the terminal. Regardless of the input method selected, processing continues in an identical manner. The list of transactions are displayed on the terminal, each preceded by an identifying number. The user is then presented with the following execution menu:

```
Pick the number or letter of the action desired
  (num) - execute one of the preceding transactions
  (d)   - redisplay the list of transactions
  (r)   - reset the currency pointer to the root
  (x)   - return to the previous menu

ACTION ----> _
```

At this point, the user can select a transaction for processing. Since each transaction is an independent entity, the order of processing is not important. The user has the option of executing any number of DL/I requests. The "r" selection causes the currency pointer to be repositioned to the root of the hierarchical schema so that subsequent requests may access the complete database, rather than be limited to beginning from a current position established by previous requests. Following each selection, the transaction is sent to the kernel mapping system (KMS) for translation, and then to the kernel controller (KC) for execution. Results, if any, are displayed to the user and the execution menu is re-displayed for further commands.

2. Query Processing in the KMS

DL/I transactions are sent to the KMS from the LIL. The function of the KMS is two-fold, to parse the request to validate the user's DL/I syntax, and to translate, or map, the request into an equivalent ABDL transaction. If the DL/I request is determined to be valid, the ABDL is passed to the kernel controller (KC) for processing and execution by the MBDS.

The primary component within the KMS is the transaction parser. It has been implemented within MM&MLDS by use of the UNIX utility Yet-Another-Compiler Compiler (YACC). YACC is a program generator that performs syntactic processing on an input stream of tokens. Given a specification of the input language structure (a set of grammar rules), the user's code to be invoked when such structures are recognized, and a low-level input routine, YACC generates a program that syntactically recognizes the input language and allows invocation of the user's code throughout this recognition process. The class of specifications accepted is a very general one: LALR(1) grammars. It is important to note that the user's code mentioned above is our mapping code that is going to perform the DL/I-to-ABDL translation. As the low-level input routine, we utilize a Lexical Analyzer Generator (LEX) [19]. LEX is a program generator designed for lexical processing of character input streams. Given a regular-expression description of the input strings, LEX generates a program that partitions the input stream into tokens and communicates these tokens to the parser.

The parser produced by YACC consists of a finite-state automaton with a stack and performs a top-down parse, with left-to-right scan and one token look-ahead. Control of the parser begins initially with the highest-level grammar rule. Control descends through the grammar hierarchy, calling lower and lower-level grammar rules which search for appropriate tokens in the input. As the appropriate tokens are recognized, some portions of the mapping code may be invoked directly. In other cases, these tokens are propagated back up the grammar hierarchy until a higher-level rule has been satisfied, at which time further translation is accomplished. When all the necessary lower-level grammar rules have been satisfied and control has ascended to the highest-level rule, the parsing and translation processes are complete. If the entire token string has been processed and associated with grammar rules, parser execution will terminate normally, otherwise, a syntax error is reported, the parser will abort, and control will return to the calling procedure.

In addition to the information provided through the data structures from the LIL, the KMS uses, primarily, five data structures during the parsing operation. These are outlined briefly for completeness. Figure 20 shows the hie_kms_info Data Structure. This structure holds information for delayed use in the parsing process. The first field in this record, tgt_list, is a pointer to the head of a list of attribute names. These are the names of those attributes whose values are retrieved from the database. The second field, insert_list, is a pointer to the head of a list of insert lists. This list generally contains a single item, which points to a single insert list. However, in the case of multiple path insertion, this list contains an item that points to each insert list, corresponding to each segment to be inserted. Each insert list, then, holds the values that an ISRT request desires inserted into the database for a given segment. The third field, temp_str, is a pointer to a variable-length character string. The character-string length is a function of the of the input request length, and is allocated, when required, to accumulate intermediate translation results while parsing the boolean predicates that optionally follow the segment name in the segment search argument (SSA) of a given user request. The final field, alt_tgt, has been added during the current implementation to hold information relating the translation to AB(network) statements.

```
struct  hie_kms_info
 {
   struct symbolic_id_info    *tgt_list;
   struct insert_lists        *insert_list;
   char                       *temp_str;
   struct alt_list_info       *alt_tgt;
 }
```

Figure 20. The hie_kms_info Data Structure

The next four data structures, shown in Figure 21, are records that are pointed to by the hie_kms_info record, as just described. Respectively they represent a list of attribute names (the target list), a list of insert list nodes, a list of attribute (the insert list(s)), and operations used in the AB(network) translations, respectively. Further details on the KMS and its data structures can be found in Benson and Wentz [11].

```
struct symbolic_id_info
 {
   char                        name[ANLength + 1];
   int                         length;
   struct  symbolic_id_info    *next_attr;
 }

struct insert_lists
 {
   char                        *list;
   int                         insert_attrs;
   int                         insert_vals;
   char                        seg_name_[RNLength + 1];
   struct  hrec_node           *seg_ptr;
   struct  insert_lists        *next_list;
 }

struct insert_info
 {
   char                        attr[ANLength + 1];
   char                        *value;
   char                        type;
   struct  insert_info         *next_val;
 }

alt_list_info
 {
   char                        name[ANLength + 1];
   char                        op[RNLength + 1];
   char                        *value;
   struct  alt_list_info       *next_attr;
 }
```

Figure 21. Additional KMS Data Structures

As described, the KMS parser is the central component in the translation of DL/I transactions to ABDL statements, hence the modification for the cross-access capability deal primarily with that construct. The code changes will principally involve branching subroutines that alter the code generation process when a grammar rule is recognized within the parser. The remaining part of this chapter describes the implementation details involved in mapping the primary DL/I transactions (GET, GET HOLD, DLET, REPL, and ISRT) to their AB(network) equivalents.

B. THE DL/I-GET TO THE ABDL-RETRIEVE

The DL/I GET statement is a combination of the CODASYL FIND and GET. The DL/I GET calls consist of the Get Unique (GU), Get Next (GN), and Get Next within Parent (GNP) operations. The fact that each of these calls is quite different in functionality is of little concern to the KMS parser/translator. All of these calls have identical form, syntactically, with the exception of the DL/I operator, i.e., GU, GN, GNP. Therefore, the KMS maps each DL/I GET call to an equivalent ABDL RETRIEVE request, or, as in most cases, a series of ABDL RETRIEVE requests. An operator identification flag is set during the translation process which allows the KC to associate the appropriate operation to these requests for controlling their execution.

The DL/I GU operation is a direct retrieval, and as such, has to specify the complete hierarchical path to the desired segment. That is, it specifies the segment type at each level of the database, from the root down to the desired segment, together with an optional occurrence-identifying condition for each segment type. (Collectively, such a specification, at each level, is referred to as a segment search argument for that level/segment.) (The DL/I GU corresponds to the CODASYL 'out of the blue' FIND.) An example of such a call is as follows:

```
GU  course (ctitle= Ada)
    offering
    student
```

This call retrieves information concerning the first occurrence of a student enrolled in the course entitled 'Ada'. The series of ABDL requests generated for such a call is as follows:

```
[ RETRIEVE  ((TEMPLATE = COURSE) and
             (CTITLE = Ada))
            (CNUM) BY CNUM ]

[ RETRIEVE  ((TEMPLATE = OFFERING) and
             (CNUM = ****))
            (DATE) BY DATE ]

[ RETRIEVE  ((TEMPLATE = STUDENT) and
             (CNUM = ****) and
             (DATE = ******))
            (SUM, SNAME, GRADE) BY SNUM ]
```

Notice that only the first RETRIEVE request generated is fully-formed, i.e., may be submitted to MBDS "as-is". Subsequent requests may not be completed until the appropriate sequence-field values have been obtained from the execution of previous requests. This process takes place in the KC. The KMS uses asterisks, as place holders, to mark the maximum allowable length of such sequence fields. Each RETRIEVE request, with the exception of the last, is generated solely to extract the hierarchical path to the desired segment. (By doing so, they allow the KC to establish and maintain the current position within each segment referenced in a DL/I call.) Consequently, the only attribute in each target list is that of the segment sequence field. Of course, the target list of the last request contains all the attributes of the desired segment. It is the information obtained from the execution of the final request which is returned to the user, via the KFS. Also of note is that each request includes the optional ABDL "By attribute_name" clause. With the implementation done by Benson and Wentz[11], the results obtained from each RETRIEVE request are sorted by sequence_field value through the inclusion of a "BY sequence_field" clause on all ABDL RETRIEVE requests.

The DL/I GN and GNP operations are sequential retrievals. As such, they may each contain a looping construct. Such a construct takes the form of a label that precedes the GN or GNP operator, and a GOTO statement following the last segment search argument of the DL/I call. GN and GNP operations are predicated on the fact that a previous DL/I call has established a current position within the database. Therefore, unlike the GU operation, they need not specify the complete hierarchical path from the root to the desired segment. This does, however, make it necessary to semantically analyze the GN or GNP, and the previous DL/I call. (GN corresponds to the simpler of the two 'relative' FIND options in CODASYL.) An example of such a call is as follows:

```
xx GNP student
   GOTO xx
```

This call retrieves information concerning the next occurrence of a student enrolled in the course, and the offering of that course, which have been established as the current COURSE and OFFERING segments within the database by the previous GET operation (of any type) or ISRT operation. The ABDL request generated for such a call is as follows:

```
[ RETRIEVE ((TEMPLATE = STUDENT) and
            (CNUM = ****) and
            (DATE = ******))
           ((SNUM, SNAME, GRADE) BY SNUM ]
```

There is no indication, from the ABDL request generated, that the DL/I call contained a looping construct. However, a loop pointer is set during the translation process which allows the KC to discern that a looping construct exists and the extent of such a construct. The KMS translator recognizes that the first segment search argument of this DL/I call does not specify the root segment as its segment type. Consequently, it performs a tree walk of the hierarchical schema, in reverse order, to obtain the sequence fields required to complete the translation process, i.e., those that specify the complete path from the root to the segment concerned; in this case, CNUM and DATE.

The GN and GNP operators may also be used to perform sequential retrieval without the specification of SSA's. In the case of the GN operator, such a call retrieves all of the segments (of all types) subordinate to the last segment type referenced in the previous DL/I call, which established the current position within the database. The GNP operator functions similarly, except that instead of retrieving all subordinate segments, it only retrieves subordinate child segments, i.e., it does not retrieve segments below the immediate children of the current parent segment. Since no SSA is specified, the KMS translator has to save the identity of the last segment type referenced in each DL/I call, i.e., which segment types constitute "subordinate" segments. An example of such a call is as follows:

```
yy GN
   GOTO yy
```

Assuming that the previous DL/I call is simply "GU course", the series of ABDL requests generated are as follows:

```
[ RETRIEVE ((TEMPLATE = PREREQ) and
            (CNUM = ****))
            (PNUM, PTITLE) BY PNUM ]

[ RETRIEVE ((TEMPLATE = OFFERING) and
            (CNUM = ****)
            (DATE, LOCATION, FORMAT) BY DATE ]

[ RETRIEVE ((TEMPLATE = TEACHER) and
            (CNUM = ****) and
            (DATE = ******))
            ((TNUM, TNAME) BY TNUM ]

[ RETRIEVE ((TEMPLATE = STUDENT) and
            (CNUM = ****) and
            (DATE = ******))
            (SNUM, SNAME, GRADE) BY SNUM ]
```

If subordinate segments for PREREQ, TEACHER or STUDENT were in the database, appropriate RETRIEVE requests would have been generated for those segment types also, i.e., PREREQ, TEACHER and STUDENT are the leaves of our example database. However, if our example DL/I call contained GNP, instead of GN, only the RETRIEVEs for PREREQ and OFFERING would be generated, i.e., only the children of COURSE in our example database. Also, notice that each request includes all attributes for that segment type in its target list. That is, complete information about each of these segments is to be returned to the user.

C. THE DL/I-GET-HOLD TO THE ABDL-RETRIEVE

The DL/I GET HOLD calls consist of the Get Hold Unique (GHU), Get Hold Next (GHN), and Get H.ld Next within Parent (GHNP) operations. A DL/I GET HOLD call is used to retrieve a given segment occurrence into a work area, and hold it there so that it may subsequently be updated or deleted. ABDL does not have this requirement. Therefore, when the KMS parser encounters one of these calls, the KMS translator treats them as a corresponding GET call. With the exception of the "H", the general form of the GET HOLD calls is identical to the forms of the non-HOLD (i.e., GET) counterparts. Thus, the mapping processes described in the previous subsection are applicable to the GET HOLD calls, with the exception of the special case of sequential retrieval without the specification of SSAs. Such a call has no meaning with a GET HOLD operator.

D. THE DL/I-DLET TO THE ABDL-DELETE

The DL/I DLET consists of a GET HOLD call, together with the reserved word DLET immediately following the last SSA in the GET HOLD portion of the call. When the KMS parser encounters the GET HOLD portion of the call, the KMS translator generates the appropriate ABDL RETRIEVE requests. Then, when the reserved word DLET is parsed, the KMS translator generates appropriate ABDL DELETE requests to delete the current segment occurrence (i.e.,for the current position just established by the GET HOLD portion of the call), as well as all of the children, grandchildren, etc. (i.e., the descendants) of the current segment occurrence. (This would correspond to the CODASYL ERASE ALL.) An example of such a call is as follows:

```
GHU course (ctitle = 'Ada')
   offering
DLET
```

Assuming that there is only one offering of the course entitled "Ada", this call deletes the occurrences of that course and offering, along with all the teachers and students associated with them. The series of ABDL requests generated for such a call is as follows:

```
[ RETRIEVE ((TEMPLATE = COURSE) and
            (CTITLE = Ada))
            (CNUM) BY CNUM ]

[ RETRIEVE ((TEMPLATE = OFFERING) and
            (CNUM = ****))
            (DATE) BY DATE ]

[ DELETE ((TEMPLATE = OFFERING) and
          (CNUM = ****) and
          (DATE = ******)) ]

[ DELETE ((TEMPLATE = TEACHER) and
          (CNUM = ****) and
          (DATE = ******)) ]

[ DELETE ((TEMPLATE = STUDENT) and
(CNUM = ****) and
(DATE = ******)) ]
```

In general, a single RETRIEVE request is generated for each SSA in the GET HOLD portion of the DL/I DLET. Then, for each segment type subordinate to the segment type referenced in the last SSA: (1) if the segment type is a leaf, a single ABDL DELETE is generated for that segment, and (2) if the segment type is not a leaf, a pair of ABDL requests are generated for that segment, one RETRIEVE and one DELETE. In our example, TEACHER and STUDENT are leaf segment types, and thus, no additional RETRIEVE requests are generated for those segment types. Notice that each RETRIEVE request simply retrieves the sequence-field attribute for the appropriate segment type. The sequence-field values are all that is required, since no information is returned to the user as a result of these RETRIEVE requests. These are the values required to complete the DELETE requests, specifying the complete hierarchical path from the root to the segment to be deleted.

E. THE DL/I-REPL TO THE ABDL-UPDATE

DL/I has been implemented in an interactive language interface. However, DL/I is an embedded database language that is normally invoked from a host language (i.e., PL/I, COBOL, or System/370 Assembler Language) by means of subroutine calls. The syntax for providing an appropriate attribute value pair to be changed during a DL/I REPL call is resident in the host language, not in the DL/I data language itself. In order to make an embedded language function interactively, the work of Benson and Wentz[11], introduced additional syntax for the language interface. This additional syntax does not represent a change to the DL/I data language, but rather, serves only to facilitate the interactive implementation of the normally embedded data language, DL/I.

Therefore, the following syntax allows the user to input the attribute-value pair they desire to change:

```
CHANGE  attribute_name  TO  attribute_value
```

The DL/I REPL consists of a GET HOLD call, with the above syntax immediately following the last SSA in the GET HOLD portion of the call, and then the reserved word REPL. When the KMS parser encounters the GET HOLD portion of the call, the KMS translator generates the appropriate ABDL RETRIEVE requests. When the KMS parser encounters the additional syntax, it saves the attribute-value pair in local variables for subsequent use by the KMS translator. Then, when the reserved word REPL is parsed, the KMS translator generates the appropriate ABDL UPDATE request to update the current segment occurrence, i.e., for the current position just established by the GET HOLD portion of the call. An example of such a DL/I REPL call is as follows:

```
GHU course (ctitle = 'Ada')
   prereq (ptitle = 'Basic')
CHANGE ptitle TO 'Pascal'
REPL
```

The series of ABDL requests for such a DL/I REPL call is as follows:

```
[ RETRIEVE ((TEMPLATE = COURSE) and
             (CTITLE = Ada))
            (CNUM) BY CNUM ]

[ RETRIEVE ((TEMPLATE = PREREQ) and
             (CNUM = ****) and
             (PTITLE = Basic))
            (PNUM) BY PNUM ]

[ UPDATE ((TEMPLATE = PREREQ) and
           (CNUM = ****) and
           (PNUM = ****))
          (PTITLE = Pascal) ]
```

Notice that each RETRIEVE request simply retrieves the sequence-field attributes for the appropriate segment type, i.e., like the DL/I DLET, no information is returned to the user as a result of these RETRIEVE requests. As is the case with ABDL, we may only update a single attribute in each DL/I REPL call. However, each DL/I REPL call updates that particular attribute-value pair in all multiple record occurrences that may exist.

F. THE DL/I-ISRT TO THE ABDL-INSERT

As in the case of the DL/I REPL, additional syntax was introduced to allow the DL/I ISRT to function in an interactive language interface. In this instance, the following syntax was implemented for the DL/I ISRT, which allows the user to build the new segment to be inserted into the database:

```
BUILD [(attr_1, ..., attr_n)] : (value_1, ..., value_n)
```

If values are to be inserted for each attribute of the segment type, there is no requirement to list the attribute names. Only the attribute values need to be listed. However, they have to appear in the same order in which they were defined during the original definition of the database. A value for the sequence-field attribute may not be omitted from the list. Due to the ABDL requirement that the INSERT request include values for all attributes, in the case where the user does not specify values for all attributes in the segment, the KMS translator inserts default value. We use a zero (0) and a "Zz" as the default values for the data types integer and character, respectively.

The DL/I ISRT consists of our additional syntax to build a new segment occurrence, followed by a sequence of SSAs, the first of which is preceded by the reserved word ISRT. This sequence of SSAs has to specify the complete hierarchical path from the root to the segment to be inserted. (The DL/I ISRT could be compared to the CODASYL STORE statement.) An example of such a call is as follows:

```
build (tnum, tname) : (1234, 'janet')
isrt course (ctitle = 'Ada')
    offering (date = 891206)
    teacher
```

The series of ABDL requests generated for such a DL/I ISRT call is as follows:

```
[ RETRIEVE ((TEMPLATE = COURSE) and
            (CTITLE = Ada))
            (CNUM) BY CNUM ]

[ RETRIEVE ((TEMPLATE = OFFERING) and
            (CNUM = ****) and
            (DATE = 891206))
            (DATE) BY DATE ]

[ INSERT ((TEMPLATE, TEACHER),
          (CNUM, ****),
          (DATE, ******),
          (TNUM, 1234),
          (TNAME, Janet)) ]
```

Although the sequence field of the OFFERING segment has been specified in its SSA, the translator does not recognize this fact. Therefore, the RETRIEVE request for the OFFERING segment is mechanically generated, in spite of the fact that we are given the value that we subsequently retrieve when executing this request. This RETRIEVE returns only one date, in this case, 891206. No RETRIEVE request is generated for the TEACHER segment. In general, no RETRIEVE request is generated for the last SSA in the DL/I ISRT. This is because the last SSA represents the segment to be inserted and, by definition, the user gives us the sequence-field value when building the new segment. The KMS translator did not have to insert any default values, as all TEACHER attributes have been listed by the user in building the new segment.

The above discussions on the role and translation processes of LI indicate that the existing LI for the hierarchical DL I interface is adequate. We merely modify the data structure of LI so that it may point to the newly created AB(network) schema.

VI. CONCLUSIONS

The predominant approach to database design has been to implement a system based on a single database model and associated data manipulation language. This has proved to be an adequate, short-term solution; however, the lack of flexibility, capacity for expansion, and extensibility indicate the need for research into alternative approaches. One such approach has been designed and implemented at the Laboratory for Database Systems Research, Naval Postgraduate School, Monterey, California. The Multi-Model and Multi-Lingual Database System (MM&MLDS), as shown in Figure 22, allows a single database system to support multiple languages and models. Specifically supported models and their associated data manipulation languages include relational/SQL, hierarchical/DL/I, network/CODASYL-DML, functional/Daplex, and attribute-based/ABDL. The system can easily be expanded to handle other database models and data manipulation languages.

Although MM&MLDS allows access and manipulation of databases via separate data models and languages, individual databases can be accessed only through the model within which it was created. For example, a network database can only be accessed via the network data model and the CODASYL-DML data manipulation language. The extension of MM&MLDS to support cross-model-accessing of all databases through any of the supported models is the current focus of research and design analysis.

The design and analysis of one of the interfaces within the MM&MLDS has been the central topic of this thesis.

A. A REVIEW OF THE RESEARCH

The goal of the research in this thesis has been to increase the functionality of MM&MLDS by allowing a database user knowledgeable only in the hierarchical model to access and manipulate a network database through the use of DL/I transactions. We presented and analyzed a number of design strategies for implementing this extension into MM&MLDS, including the high-level preprocessing, mixed-processing, and postprocessing methods before selecting the mixed-processing as the most feasible methodology.

In order to implement the mixed-processing strategy, two components, a schema transformer and a new language interface, had to be designed. We first discussed the design of a schema transformation algorithm from a network database to a hierarchical

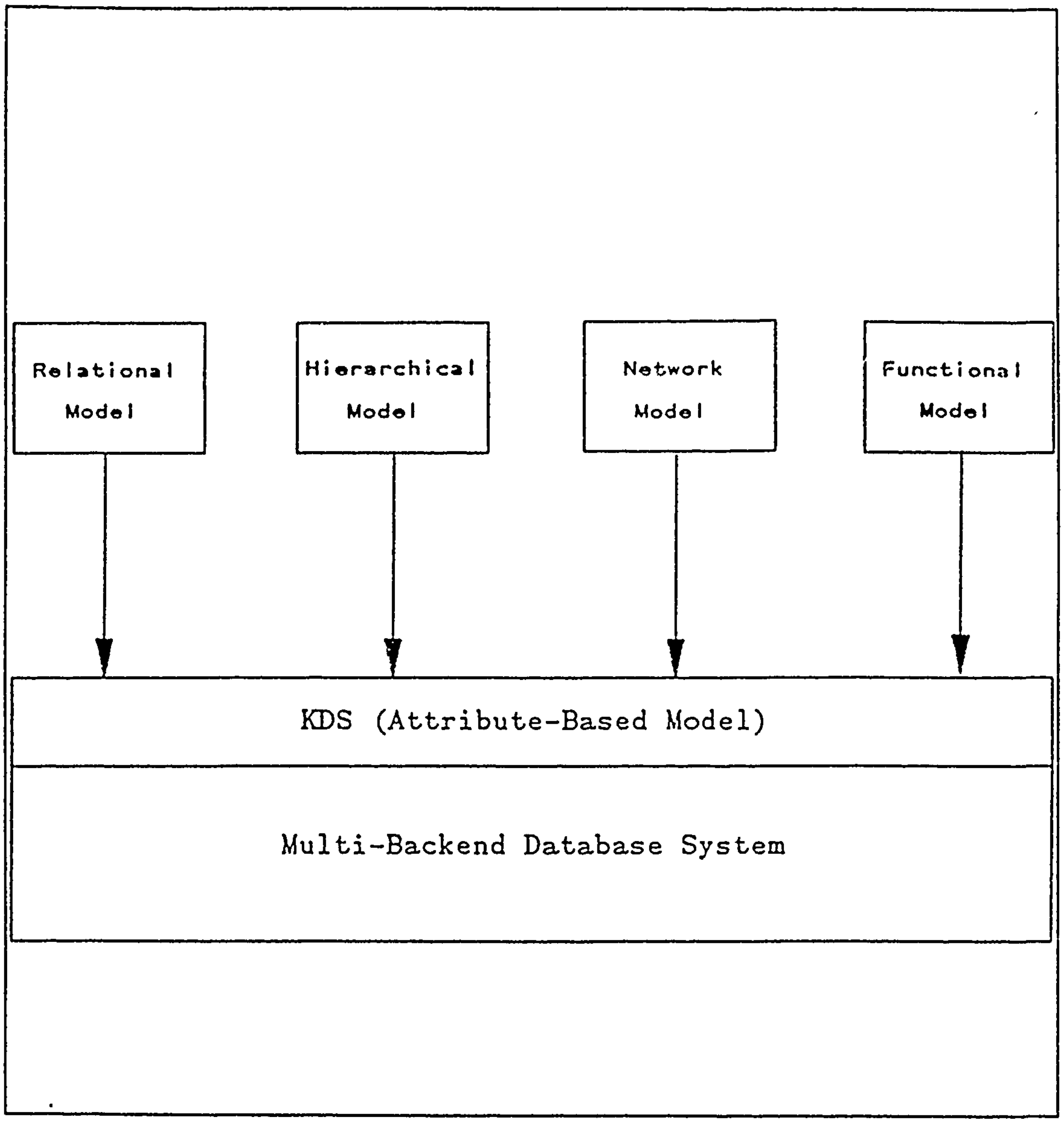


Figure 22. The Multi-Model and Multi-Lingual Database System Concept

database. The technique selected involved breaking the *many-to-many* relationships of the network structure into several *one-to-many* hierarchies and the cascading of data in key fields in network records into the hierarchical schema to fully codify the owner-member relationships among records of network sets. We then described the data structures and implementation details necessary to integrate the schema transformer into the Language Interface Layer (LIL).

The design and analysis of the new language interface provides the means for accessing and manipulating a network database by the translation of DL/I statements to their AB(network) equivalents. We discussed how it was possible to create a new, 'logical' language interface by modification of the existing hierarchical language interface and incorporation of the functionality of the network language interface into a framework that provides the cross-accessing capability. We then discussed the modification necessary to the hierarchical KMS and KC to implement the new language interface, and concluded by describing the translations of the DL/I statements to the equivalent AB(network) transactions.

B. FINAL OBSERVATIONS

Figure 23 depicts the Cross-Model Accessing concept as a functional extension of MM&MLDS. Earlier design [17] and implementation[24] of the capability to access a functional database using the network model and the CODASYL-DML data manipulation language, and the design and implementation of the capability to access a hierarchical database using the relational model and the SQL data manipulation language[18]; along with the work done in this thesis, support and confirm the feasibility of the MM&MLDS design. Additionally, this body of research provides the basis for designing alternate cross-model accessing capabilities.

Currently planned thesis topics on the Multi-Model and Multi-Lingual Database System include additional cross-model accessing capabilities. The ongoing research and development efforts at the Laboratory for Database Research indicate that a fully-functional multi-model and multi-lingual database system can be designed and implemented utilizing current hardware and software techniques and that additional growth capacity, performance gains, and extensibility of such a system is a significant step in the area of database design.

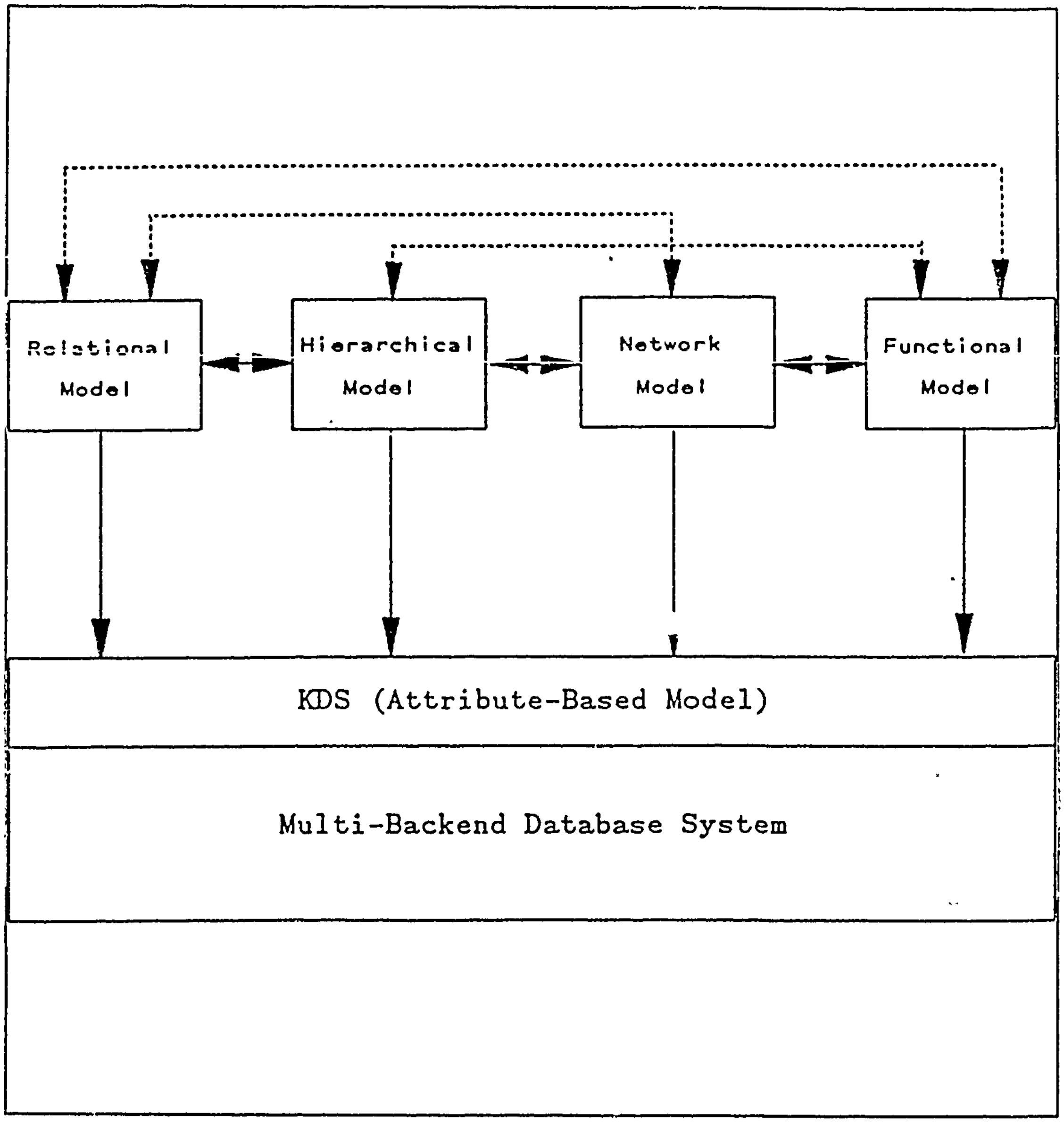
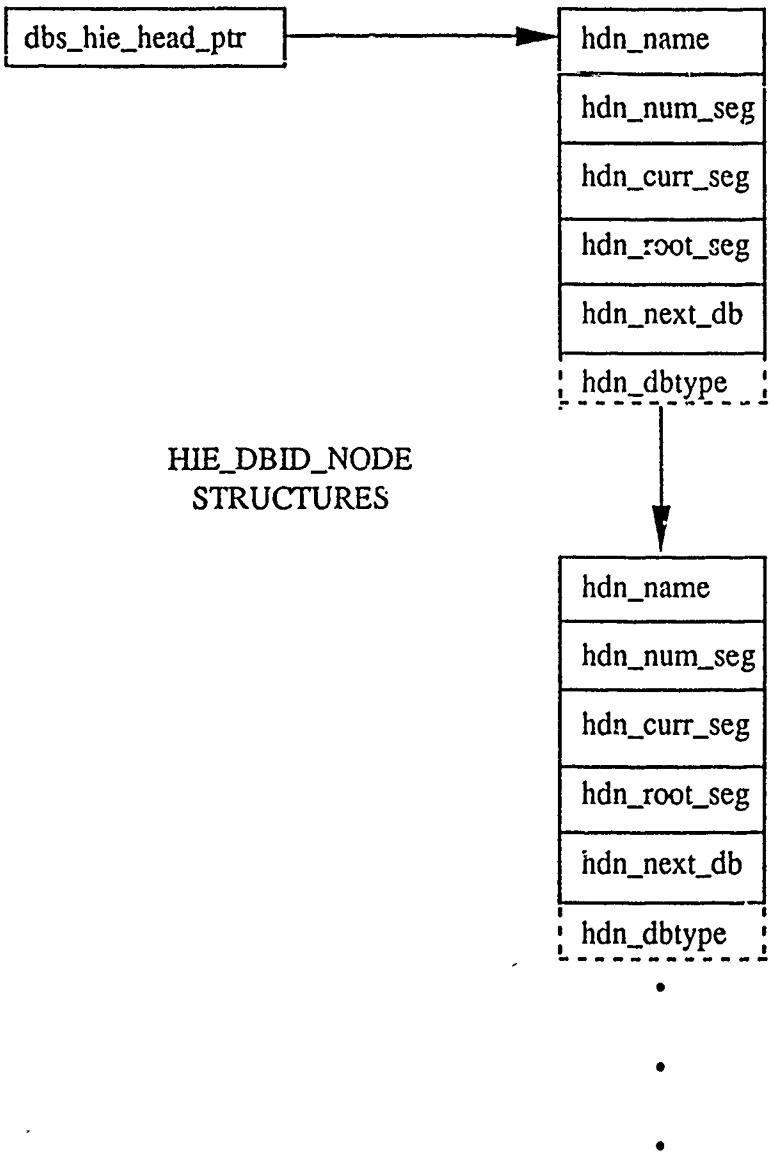
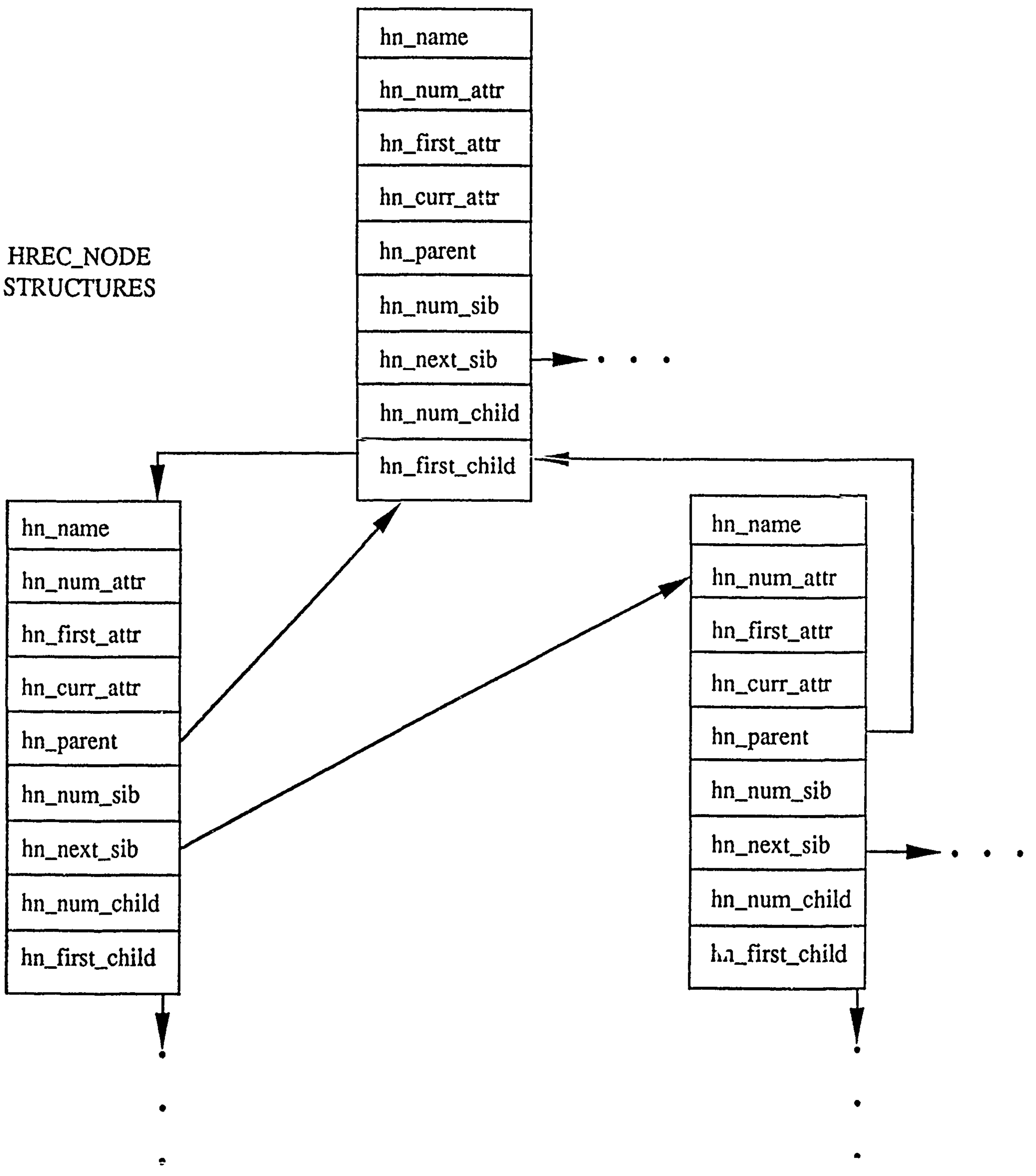


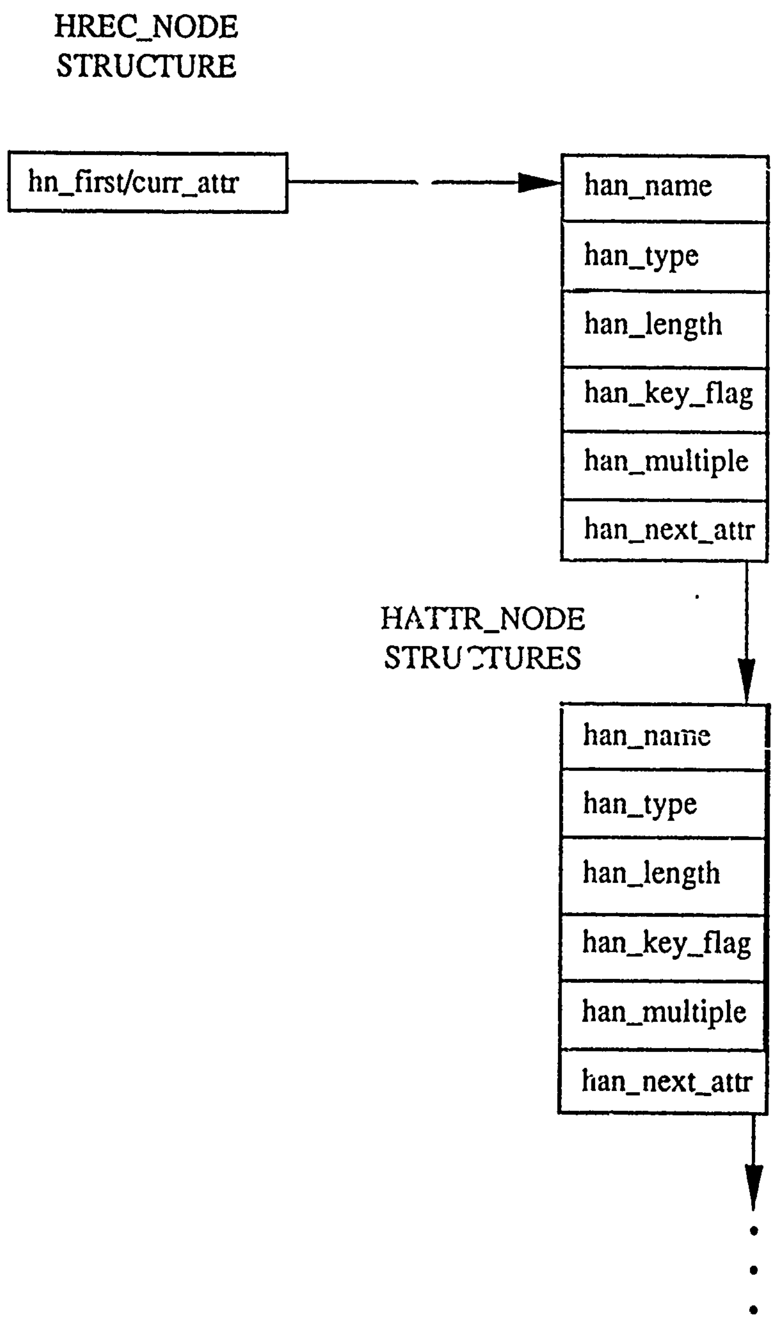
Figure 23. The MM&MLDS With Cross-Model Accessing Concept

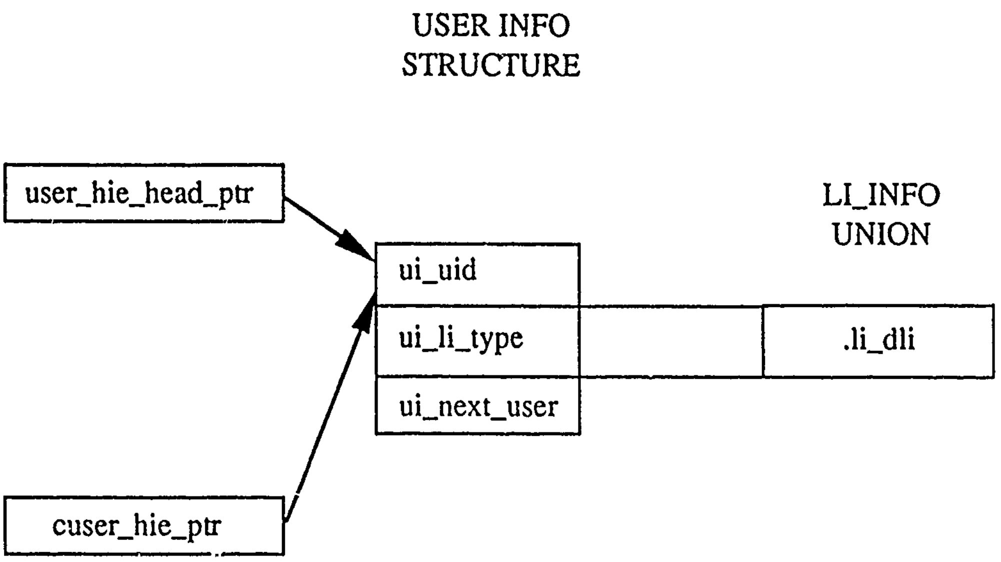
APPENDIX A. SCHEMATIC OF THE MAJOR DATA STRUCTURES

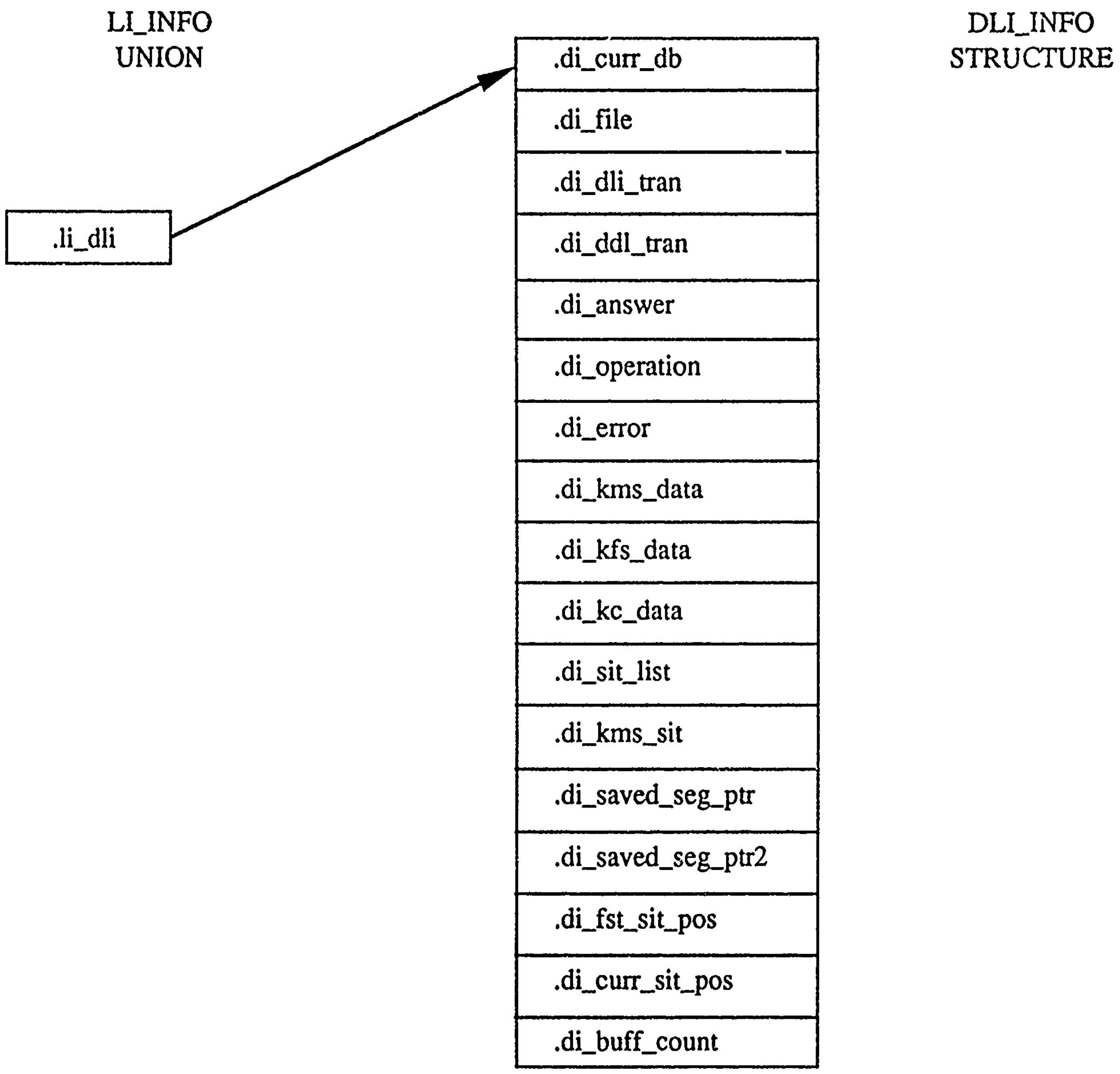


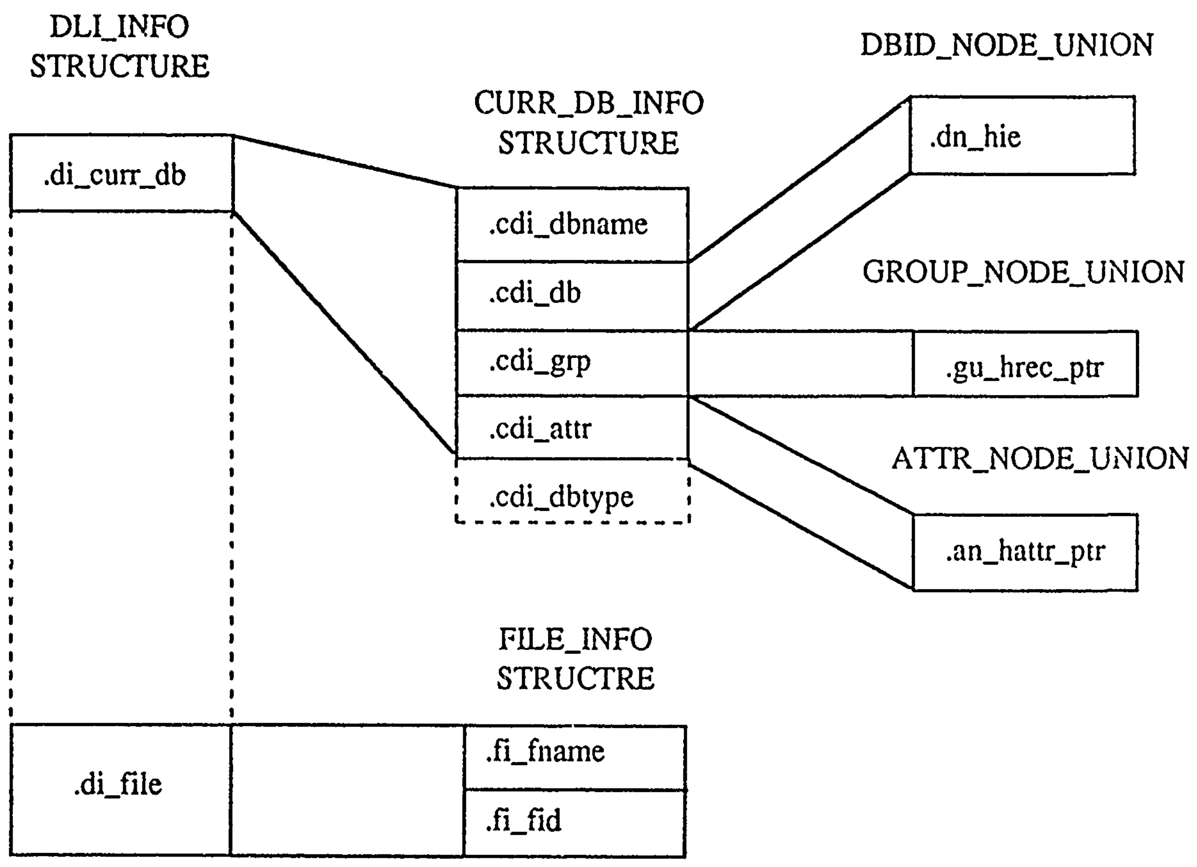
HREC_NODE
STRUCTURES

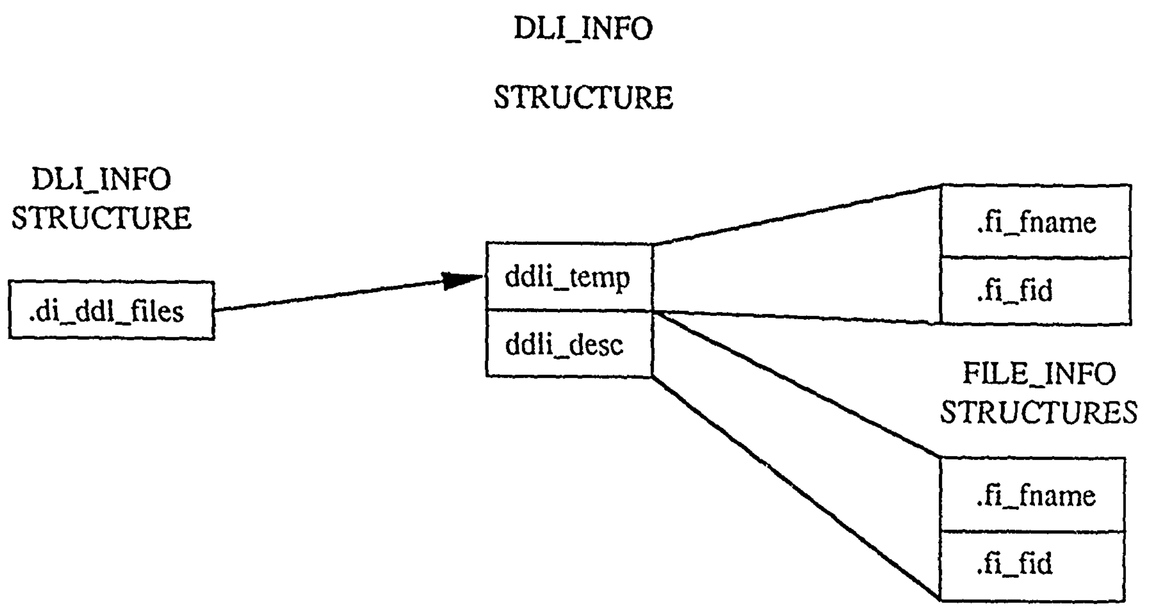


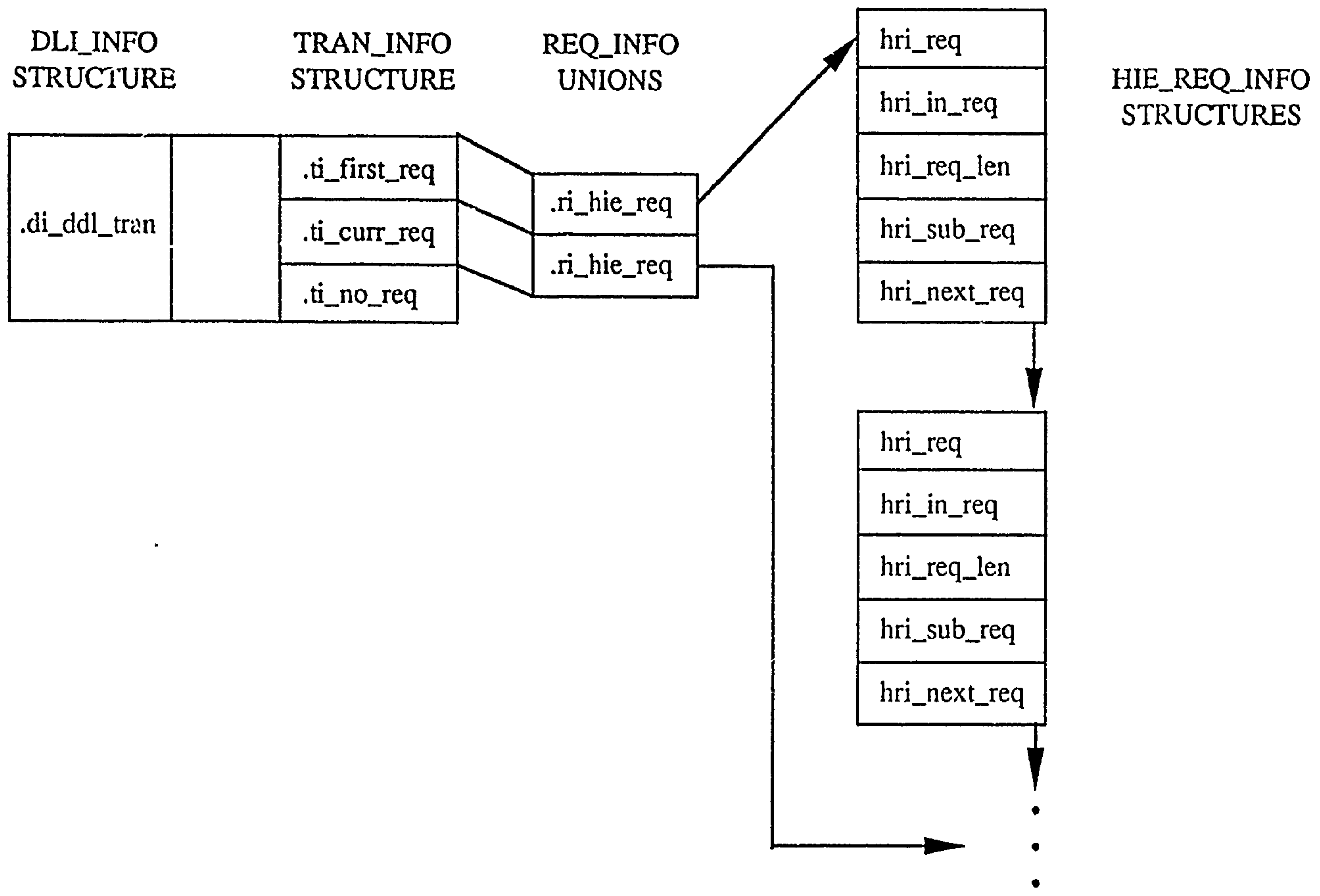

HREC_NODE
STRUCTURE
hn_first/curr_attr
han_name
han_type
han_length
han_key_flag
han_multiple
han_next_attr
HATTR_NODE
STRUCTURES
han_name
han_type
han_length
han_key_flag
han_multiple
han_next_attr

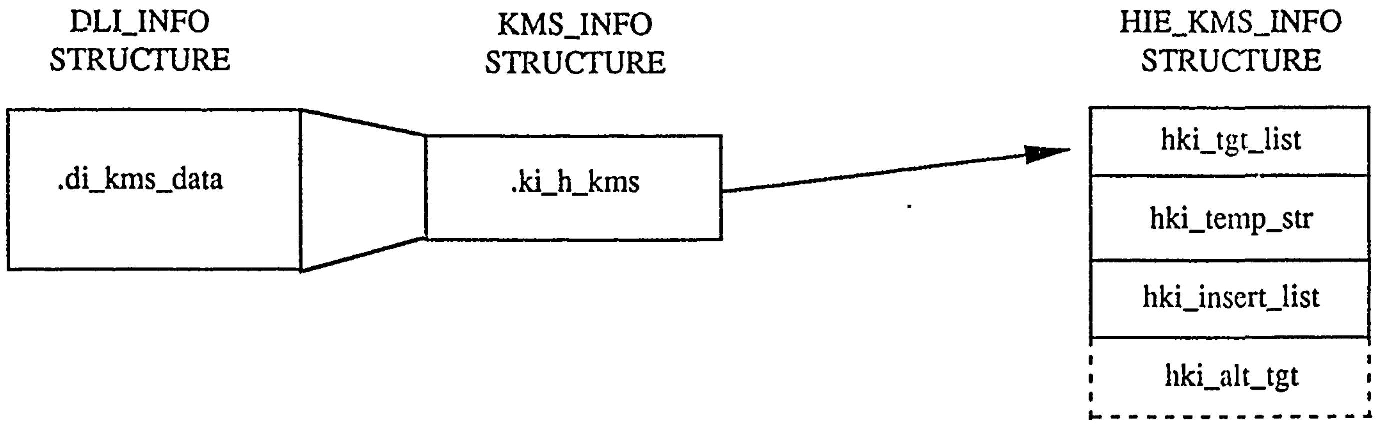

USER INFO STRUCTURE
user_hie_head_ptr
ui_uid
ui_li_type
ui_next_user
cuser_hie_ptr
LI_INFO UNION
.li_dli

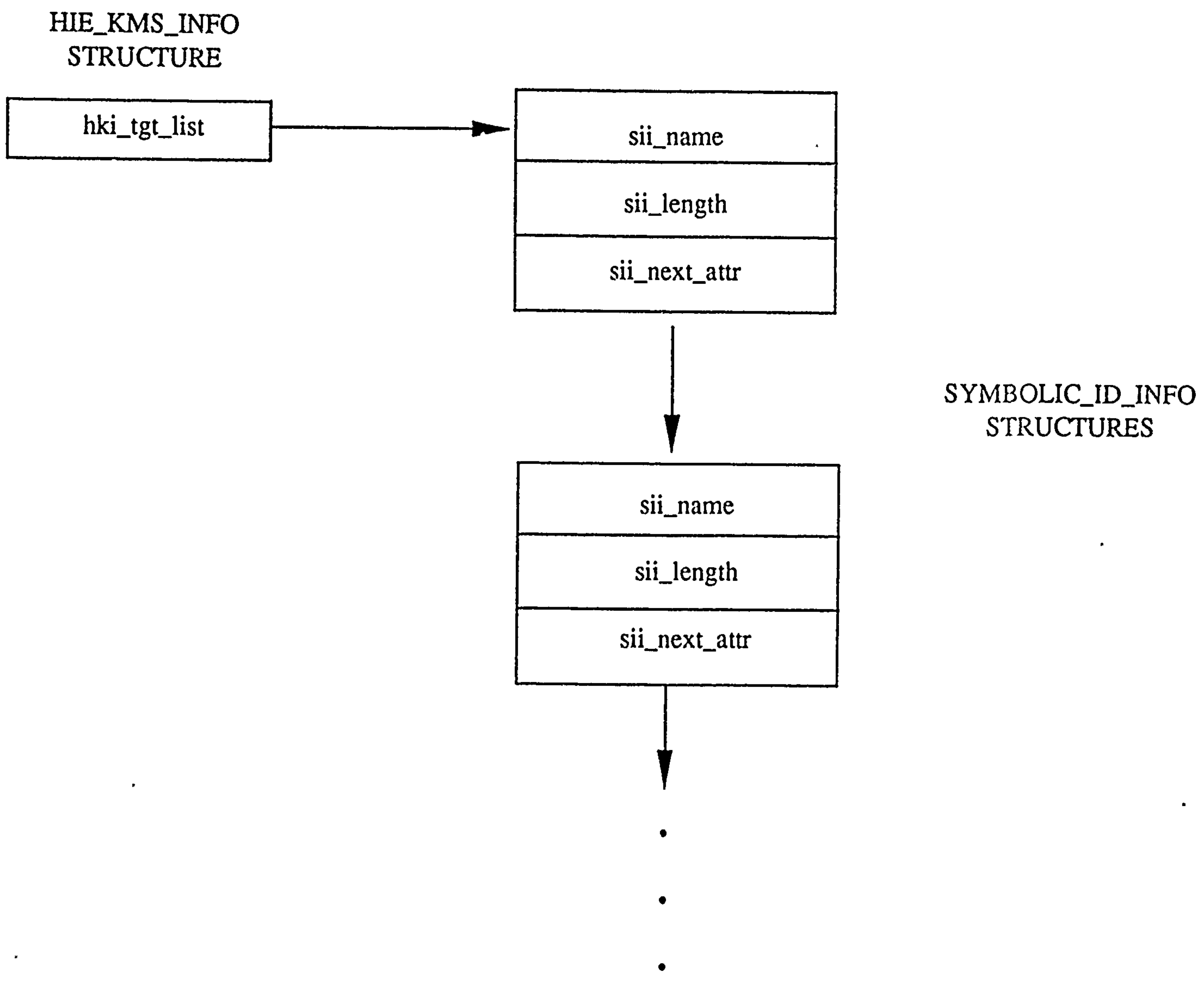

LI_INFO
UNION
.li_dli
DLI_INFO
STRUCTURE
.di_curr_db
.di_file
.di_dli_tran
.di_ddl_tran
.di_answer
.di_operation
.di_error
.di_kms_data
.di_kfs_data
.di_kc_data
.di_sit_list
.di_kms_sit
.di_saved_seg_ptr
.di_saved_seg_ptr2
.di_fst_sit_pos
.di_curr_sit_pos
.di_buff_count

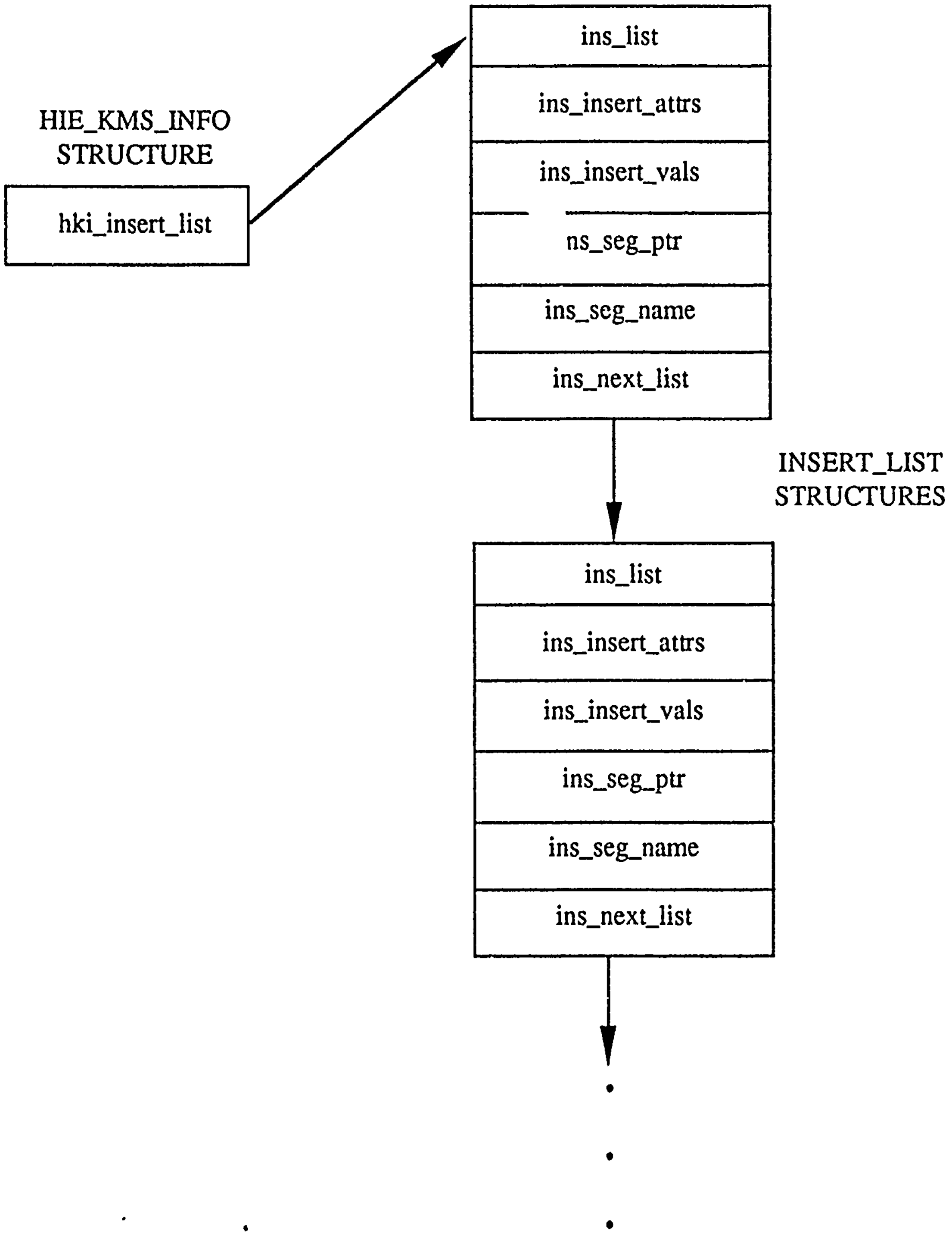

DLI_INFO STRUCTURE
.di_curr_db
.di_file
CURR_DB_INFO STRUCTURE
.cdi_dbname
.cdi_db
.cdi_grp
.cdi_attr
.cdi_dbtype
DBID_NODE_UNION
.dn_hie
GROUP_NODE_UNION
.gu_hrec_ptr
ATTR_NODE_UNION
.an_hattr_ptr
FILE_INFO STRUCTRE
.fi_fname
.fi_fid

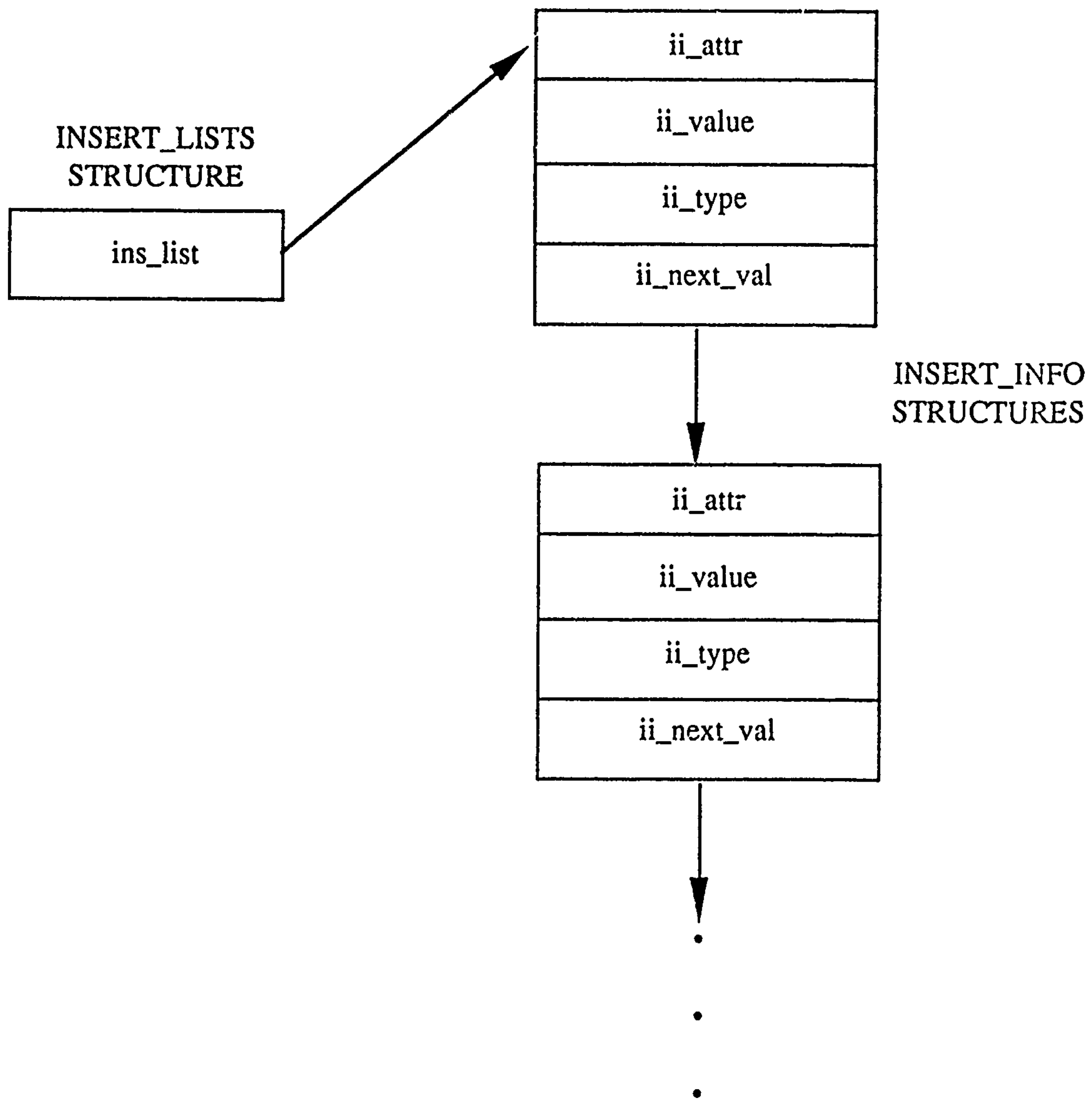

DLI_INFO
STRUCTURE
DLI_INFO
STRUCTURE
.di_ddl_files
ddli_temp
ddli_desc
.fi_fname
.fi_fid
FILE_INFO
STRUCTURES
.fi_fname
.fi_fid

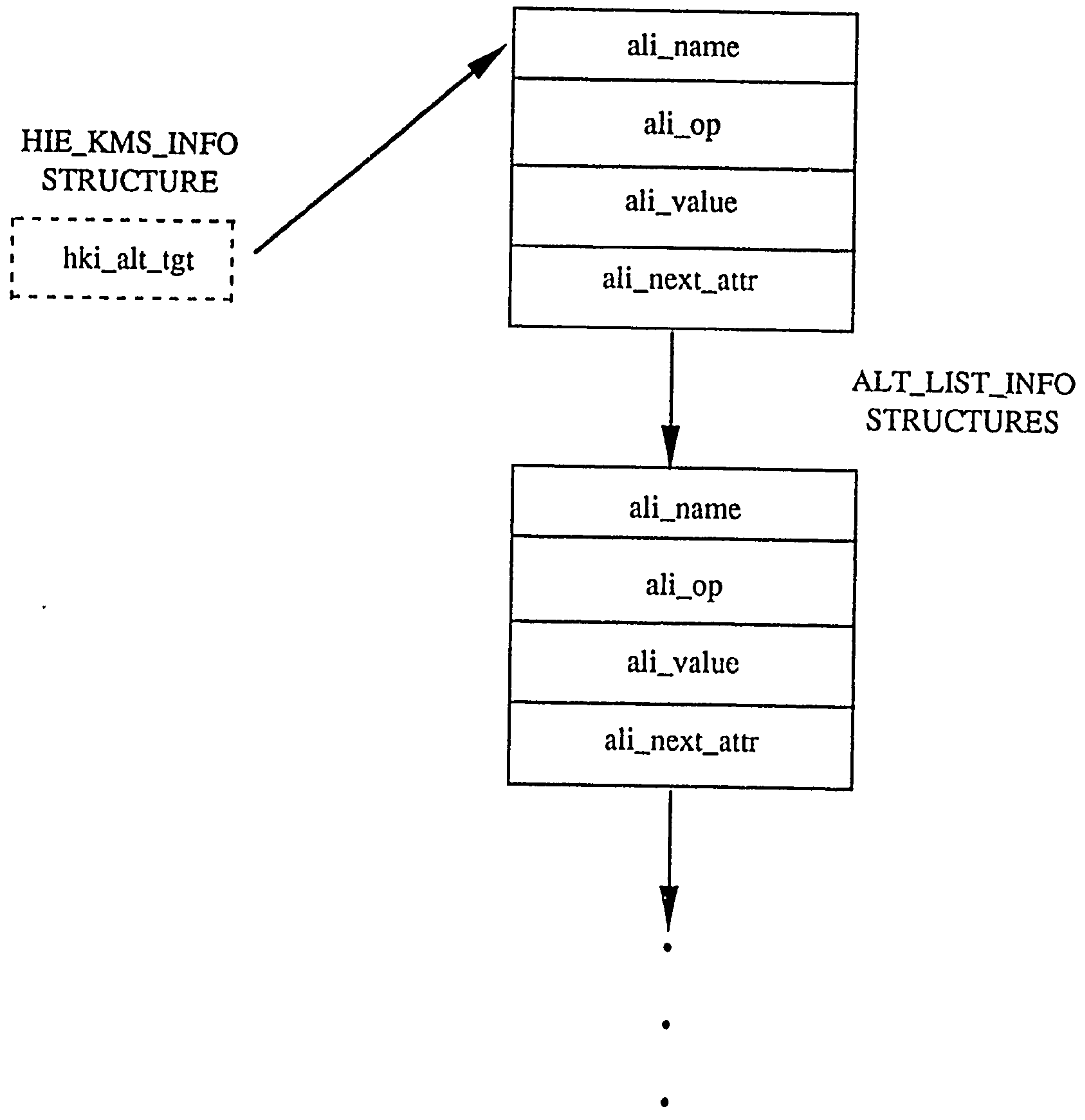

DLI_INFO STRUCTURE
TRAN_INFO STRUCTURE
REQ_INFO UNIONS
HIE_REQ_INFO STRUCTURES
.di_ddl_tran
.ti_first_req
.ti_curr_req
.ti_no_req
.ri_hie_req
.ri_hie_req
hri_req
hri_in_req
hri_req_len
hri_sub_req
hri_next_req
hri_req
hri_in_req
hri_req_len
hri_sub_req
hri_next_req

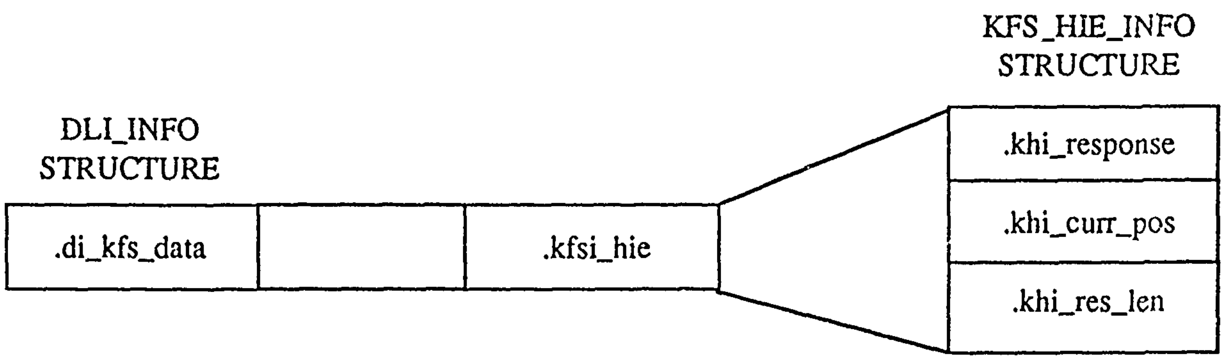

DLI_INFO STRUCTURE
.di_kms_data
KMS_INFO STRUCTURE
.ki_h_kms
HIE_KMS_INFO STRUCTURE
hki_tgt_list
hki_temp_str
hki_insert_list
hki_alt_tgt

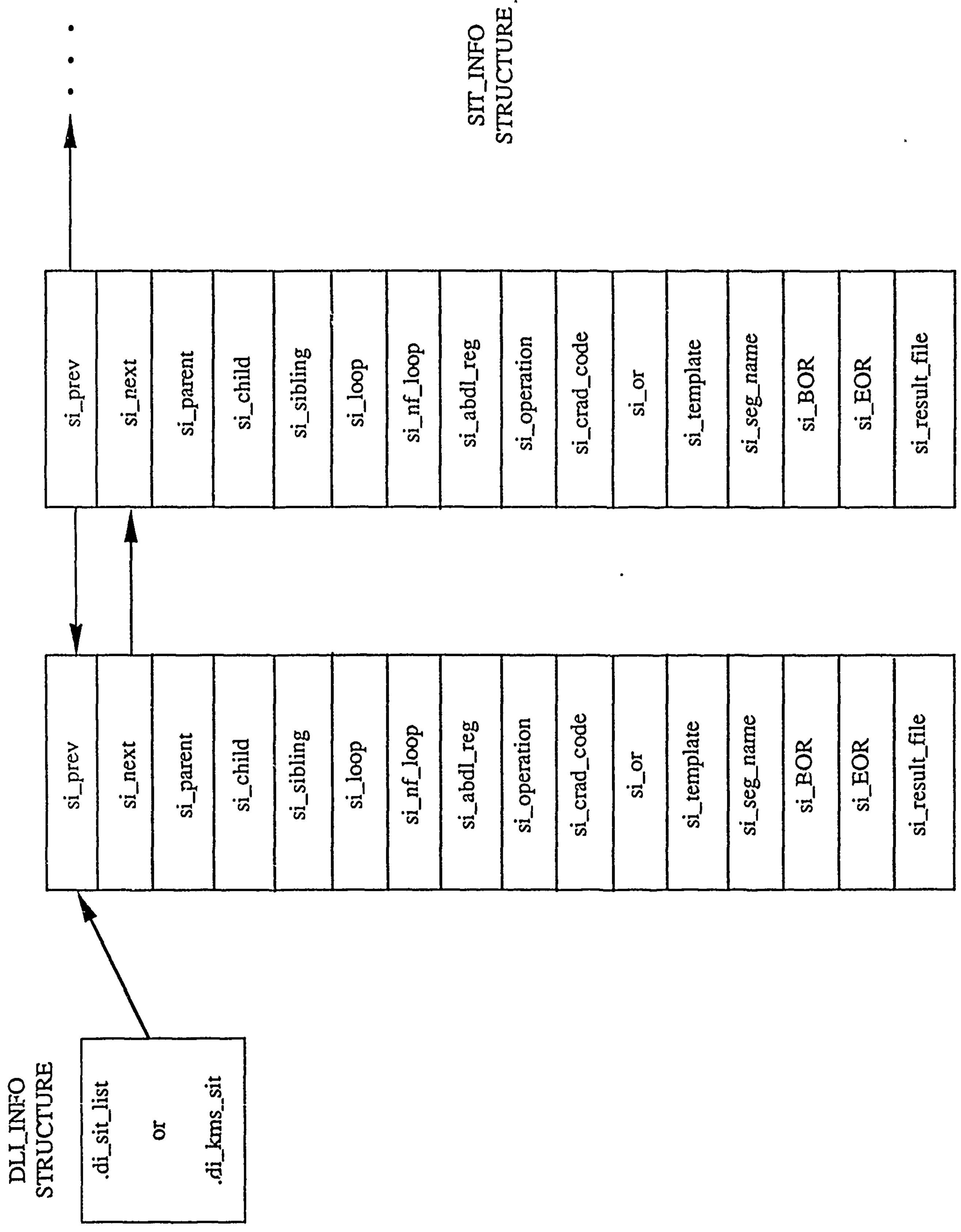

HIE_KMS_INFO STRUCTURE
hki_tgt_list
sii_name
sii_length
sii_next_attr
SYMBOLIC_ID_INFO STRUCTURES
sii_name
sii_length
sii_next_attr

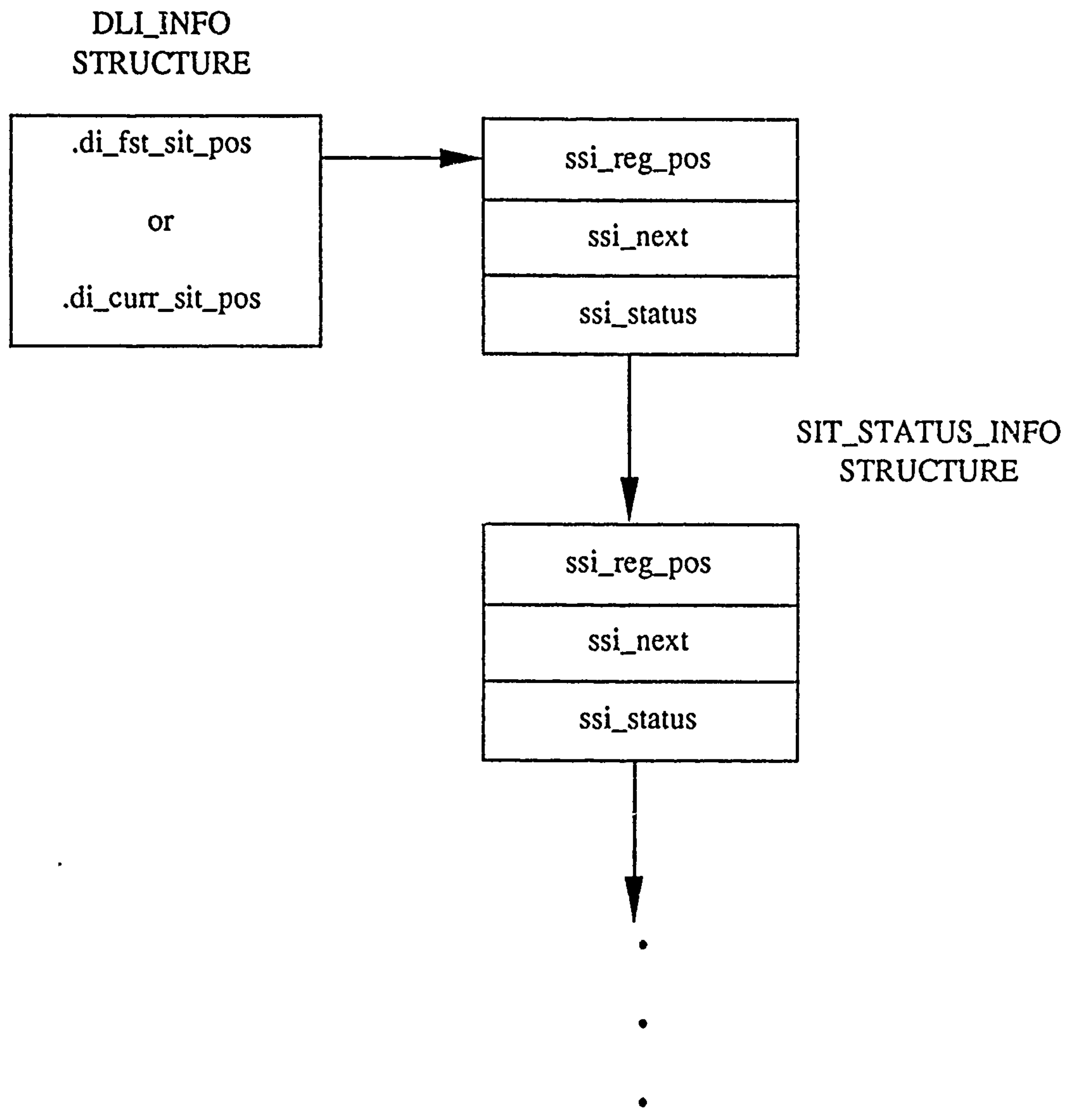

HIE_KMS_INFO STRUCTURE
hki_insert_list
ins_list
ins_insert_attrs
ins_insert_vals
ns_seg_ptr
ins_seg_name
ins_next_list
INSERT_LIST STRUCTURES
ins_list
ins_insert_attrs
ins_insert_vals
ins_seg_ptr
ins_seg_name
ins_next_list

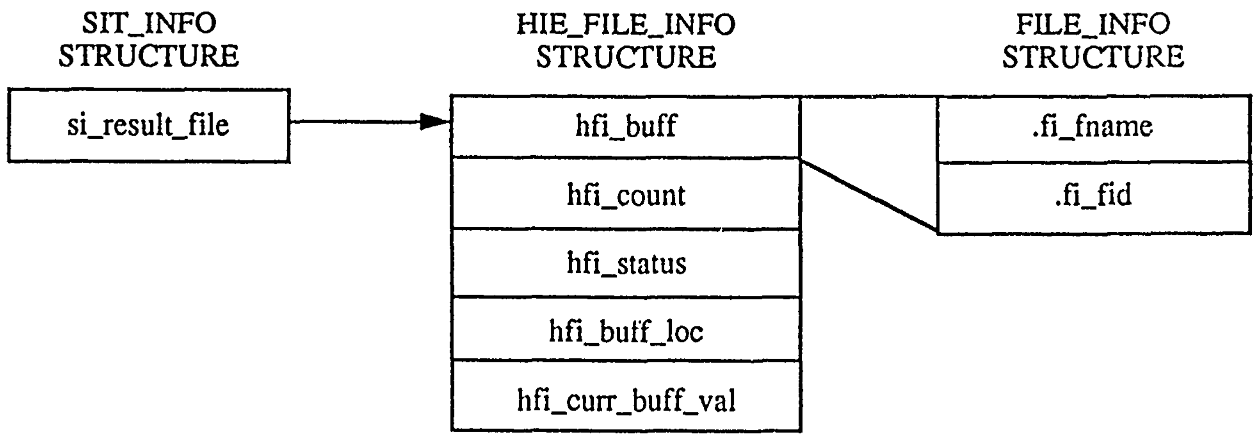

INSERT_LISTS STRUCTURE
ins_list
ii_attr
ii_value
ii_type
ii_next_val
INSERT_INFO STRUCTURES
ii_attr
ii_value
ii_type
ii_next_val


HIE_KMS_INFO STRUCTURE
hki_alt_tgt
ali_name
ali_op
ali_value
ali_next_attr
ALT_LIST_INFO STRUCTURES
ali_name
ali_op
ali_value
ali_next_attr


KFS_HIE_INFO STRUCTURE
DLI_INFO STRUCTURE
.di_kfs_data
.kfsi_hie
.khi_response
.khi_curr_pos
.khi_res_len


DLI_INFO STRUCTURE
.di_sit_list
or
.di_kms_sit
si_prev
si_next
si_parent
si_child
si_sibling
si_loop
si_nf_loop
si_abdl_reg
si_operation
si_crad_code
si_or
si_template
si_seg_name
si_BOR
si_EOR
si_result_file
si_prev
si_next
si_parent
si_child
si_sibling
si_loop
si_nf_loop
si_abdl_reg
si_operation
si_crad_code
si_or
si_template
si_seg_name
si_BOR
si_EOR
si_result_file
SIT_INFO STRUCTURE


DLI_INFO STRUCTURE
.di_fst_sit_pos
or
.di_curr_sit_pos
ssi_reg_pos
ssi_next
ssi_status
SIT_STATUS_INFO STRUCTURE
ssi_reg_pos
ssi_next
ssi_status


SIT_INFO STRUCTURE
si_result_file
HIE_FILE_INFO STRUCTURE
hfi_buff
hfi_count
hfi_status
hfi_buff_loc
hfi_curr_buff_val
FILE_INFO STRUCTURE
.fi_fname
.fi_fid

APPENDIX B. THE LIL PROGRAM SPECIFICATIONS

Note : italicized lines indicate Cross-access modifications.

```
module DLI-INTERFACE

  db-list: list;      /* list of existing hierarchical schemas  */
  head-db-list-ptr: ptr; /* ptr to head of the hierarchical schema list*/
  current-ptr: ptr;      /* ptr to the current db schema in the list */
  follow-ptr: ptr;       /* ptr to the previous db schema in the list */
  db-id: string;         /* string that identifies current db in use */


proc LANGUAGE-INTERFACE-LAYER();
     /* This proc allows the user to interface with the system. */
     /* Input and output : user DLI requests    */

     stop : int;  /* boolean flag */
     answer : char;  /* user answers to terminal prompts */

perform DLI-INIT();
stop = 'false';
while (not stop) do
  /* allow user choice of several processing operations */
  print ("Enter type of operation desired");
  print ("  (l) - load new database");
  print ("  (p) - process existing database");
  print ("  (x) - return to the operating system");
  read (answer);

  case (answer) of
   'l': /* user desires to load a new database */
         perform LOAD-NEW ();
   'p': /* user desires to process an existing database */
         perform PROCESS-OLD ();
   'x': /* user desires to exit to the operating system */
        /* database list must be saved back to a file    */
        store-free-db-list (head-db-list, db-list);
        stop = 'true';
        exit();

   default: /* user did not select a valid choice from the menu */
         print ("Error - invalid operation selected");
         print ("Please pick again");
  end-case;

  /* return to main menu */
end-while;

end-proc;
```

```
proc DLI-INIT();


end-proc;


proc LOAD-NEW();
 /* This proc accomplishes the following:                      */
 /* (1) determines if the new database name already exists,          */
 /* (2) adds a new header node to the list of schemas,      */
 /* (3) determines the user input mode (file/terminal),               */
 /* (4) reads the user input and forwards it to the parser, and       */
 /* (5) calls the routine that builds the template/descriptor files */

 answer: int;      /* user answer to terminal prompts  */
 more-input: int; /* boolean flag  */
 proceed: int;     /* boolean flag  */
 stop: int;      /* boolean flag  */
 db-list-ptr: ptr;  /* pointer to the current database  */
 req-str: str;      /* single create in DLI form  */
 ptr-abdl-list: ptr; /* ptr to a list of ABDL queries (nil for this proc)*/
 tfid,dfid: ptr;  /* pointers to the template and descriptor files */


/* prompt user for name of new database */
print ("Enter name of database");
readstr (db-id);
db-list-ptr = head-db-list-ptr;

stop = 'false';
while (not stop) do
 /* determine if new database name already exists */
 /* by traversing list of hierarchical db schemas */
 if (db-list-ptr.db-id = existing db) then
    print ("Error - db name already exists");
    print ("Please reenter db name");
    readstr (db-id);
    db-list-ptr = head-db-list-ptr;
 end-if;
 else
  if (db-list-ptr + 1 = 'nil') then
    stop = 'true';
  else
    /* increment to next database */
    db-list-ptr = db-list-ptr + 1;
 end-else;


 end-while;
```

```
/* continue - user input a valid 'new' database name */
/* add a new header node to the list of schemas and fill-in db name */
/* append new header node to db-list */
create-new-db(db-id);

/* the KMS takes the DLI defines and builds a new list of relations */
/* for the database.  After all of the defines have been processed */
/* the template and descriptor files are constructed by traversing */
/* the new database definition (schema).*/

more-input = 'true';
while (more-input) do
 /* determine user's mode of input */
 print ("Enter mode of input desired");
 print ("  (f) - read in a group of defines from a file");
 print ("  (x) - return to the main menu");
 read (answer);

 case (answer) of
  'f': /* user input is from a file */
       perform READ-TRANSACTION-FILE();
       perform DBD-TO-KMS();
       perform FREE-REQUESTS();
       perform BUILD-DDL-FILES();
       perform KERNEL-CONTROI ER();

  'x': /* exit back to LIL */
       more-input = 'false';

  default: /* user did not select a valid choice from the menu */
      print ("Error - invalid input mode selected");
      print ("Please pick again");
 end-case;
end-while;

end-proc;
```

```
proc PROCESS-OLD();
  /* This proc accomplishes the following:                   */
  /* (1) determines if the database name already exists,    */
  /* (2) determines the user input mode(file/terminal),      */
  /* (3) reads the user input and forwards it to the parser  */

  answer: int;    /* user answer to terminal prompts */
  found: int;     /* boolean flag to determine if db name is found */
  more-input: int;/* boolean flag to return user to LIL */
  proceed: int;    /* boolean flag to return user to mode menu */
  db-list-ptr:ptr;  /* pointer to the current database */
  req-str:str;  /* single query in DLI form */
  ptr-abdl-list:ptr;  /* pointer to a list of queries in ABDL form */
  tfid,dfid:ptr; /* pointers to the template and descriptor files */


 /* prompt user for name of existing database */
 print ("Enter name of database");
 readstr (db-id);
 db-list-ptr = head-db-list-ptr;


 found = 'false';
 while (not found) do
  /* determine if database name does exist      */
  /* by traversing list of hierarchical schemas */
  if (db-id = existing db) then
    found = 'true';
  end-if
  else
  /* check if db name is defined in another model */
  perform CHECK-ALTERNATE-MODELS();
  else
    db-list-ptr = db-list-ptr + 1;
     /* error condition causes end of list ('nil') to be reached */
     if (db-list-ptr = 'nil') then
       print ("Error - db name does not exist");
       print ("Please reenter valid db name");
       readstr (db-id);
       db-list-ptr = head-db-list-ptr;
     end-if;

  end-else;

 end-while;
```

```
/* continue - user input a valid existing database name */
/* determine user's mode of input */

more-input = 'true';
while (more-input) do
 print ("Enter mode of input desired");
 print ("  (f) - read in a group of DL/I requests from a file");
 print ("  (t) - read in a single DL/I request from the terminal");
 print ("  (x) - return to the previous menu");
 read (answer);

 case (answer) of
  'f': /* user input is from a file */
       perform READ-TRANSACTION-FILE();
       perform DLIREQS-TO-KMS();
       perform FREE-REQUESTS();

  't': /* user input is from the terminal */
       perform READ-TERMINAL
       perform DLIREQS-TO-KMS();
       perform FREE-REQUESTS();

  'x': /* user wishes to return to LIL menu */
       more-input = 'false';

  default: /* user did not select a valid choice from the menu */
       print ("Error - invalid input mode selected");
       print ("Please pick again");
 end-case;

end-while;

end-proc;


proc READ-TRANSACTION-FILE();
  /* This routine opens a dbd/request file and reads the transactions */
  /* into the transactions list.  If open file fails, loop until valid */
  /* file entered                                                      */

  while (not open file) do
   print ("Filename does not exist");
   print ("Please reenter a valid filename");
   readstr (file);
  end-while;

  READ-FILE();

end-proc;
```

```
proc READ-FILE();
    /* This routine reads transactions from either a file or the */
    /* terminal into the user's request list structure so that   */
    /* each request may be sent to the KERNEL-MAPPING-SYSTEM.    */

end-proc;


proc READ-TERMINAL();
    /* This routine substitutes the STDIN filename for the read   */
    /* command so that input may be intercepted from the terminal */

end-proc;


proc DBD-TO-KMS();
    /* This routine sends the request list of database descriptions */
    /* one by one to the KERNEL-MAPPING-SYSTEM*/

    while (more-dbds) do
     KERNEL-MAPPING-SYSTEM();
    end-while;

end-proc;


proc CHECK-ALTERNATE-MODELS();
  /* This routine calls other subroutines that check Relational,    */
  /* Network, and Functional schemas for the desired database name. */
  /* If found, the schema is translated to a corresponding          */
  /* hierarchical schema and prepared for processing.               */

  perform TRAVERSE-NETWORK-SCHEMA();
  if found == true
    dbtype = NET;
    perform TRANSLATE-NET-TO-HIE()
    /* initialize the database */
    hie_operation = CreateDB;
    Kernel_Controller();
 if found == false
     {
     /* stub for future implementation of functional model */
     }
    /* end check_alternate_models */
end-proc;
```

```
proc TRAVERSE-NET-SCHEMA();
   /* This routine traverses the linked list of network          */
   /* database schemas in an attempt to locate the user-requested */
   /* database.  If found, a pointer is returned to the schema.   */

end-proc;


proc TRANSLATE-NET-TO-HIE();

  /* This routine converts the network schema to hierarchical schema */

  /* The new hie database node is allocated and filled here with   */
  /* information from the network database node
  create new hie_dbid_node();
  strncpy(hierarchical_db_name, network_db_name);
  number_of_segments = number_of_records;
  dbtype = NET; /* identify db as network  */
  previous_node_next_db = new_dbid_node;  /* connect to hie db list */
```

```
proc DLIREQS-TO-KMS();
  /* This routine causes the DL/I requests to be listed to the screen. */
  /* The selection menu is then displayed allowing any of the          */
  /* DL/I requests to be executed.*/

  perform LIST-DLIREQS();
  proceed = 'true';
  while (proceed) do
   print ("Pick the number or letter of the action desired");
   print ("   (num) - execute one of the preceding DL/I requests");
   print ("   (d)   - redisplay the file of DL/I requests");
   print ("   (r)   - reset currency pointer to the root");
   print ("   (x)   - return to the previous menu");
   read (answer);

   case (answer) of
    'num' : /* execute one of the requests  */
         traverse query list to correct query;
         perform KERNEL-MAPPING-SYSTEM();
         perform KERNEL-CONTROLLER();

    'd'   :/* redisplay requests  */
 perform LIST-DLIREQS();

    'r': /* reset currency ptr to the root  */
         perform CURR-PTR-TO-ROOT();

    'x': /* exit to mode menu  */
         proceed  = 'false';

    default : /* user did not select a valid choice from the menu */
         print ("Error - invalid option selected");
         print ("Please pick again");
   end-case;

  end-while;

end-proc;
```

APPENDIX C. THE KMS PROGRAM SPECIFICATIONS

```
proc kernel-mapping-system()
 perform parser()
 perform match()
end-proc kernel-mapping-system


proc parser()
 if (operation != CreateDB, vice work with existing DB)
   alloc and init initial kms data structures
   access and save length of dli request
   free any existing abdl-str(s) from a previous parse
 end-if

 initialize the input request ptr
 perform yyparse()
 reset all booleans and counter variables


 if (operation != CreateDB)
   free all kms-unique data structures
 end-if
end-proc parser


proc yyparse()

  /* This proc accomplishes the following:*/
  /* (1) parses the DLI input requests and maps them to appropriate */
  /*     abdl requests, using LEX and YACC to build proc yyparse(), */
  /* (2) builds the hierarchical schema, when loading a new db,*/
  /* (3) checks for validity of segment and attribute names within  */
  /*     the given db schema, when processing requests against an   */
  /*     existing db.*/
```

```
%{
   boolean: creating  /* signals a DBLoad vs a DBQuery */
   boolean: updating  /* signals a DLI update request */
   boolean: label     /* signals DLI statement has a label */
   boolean: or-where  /* signals an OR term in SSA predicate */
   boolean: and-where /* signals an AND term in SSA predicate */
   boolean: literal-const /* signals alpha constant vs integer constant*/
   boolean: inserting  /* signals ISRT operation */
   boolean: not-marked  /* label not marked for attachment of loop ptr */
   boolean: first-ssa  /* signals working on 1st ssa of DLI req */
   boolean: seq-fld-has-value /* the 'value' of seq-fld is given in req*/
   boolean: missing-root  /* missing root seg in ssa specif of DLI req */
   boolean: goto-found  /* GOTO found following last ssa in DLI req */
   boolean: spec-ret-op  /* special retrieve op (GN or GNP--all segs) */
   boolean: single-build /* single segment to be built for ISRT op */
   boolean: star-d  /* cmd code D used in DLI ISRT op */
   ptr: seg-ptr  /* ptr to a schema segment */
   ptr: curr-seg-ptr  /* ptr to current segment in schema */
   ptr: prev-seg-ptr  /* ptr to previous segment in schema */
   ptr: parent-ptr  /* ptr to parent of current segment in schema */
   ptr: curr-1st-child-ptr /* ptr to 1st child of curr parent in schema*/
   ptr: label-ptr    /* ptr to abdl-str that correspondsto label */
   int: attr-len  /*length of current attribute */
   int: insert-attrs  /* number of attrs inserted during ISRT op */
   int: insert-vals   /* number of vals inserted during ISRT op */
   int: operator-flag  /* dli-operator in DLI request */
   int: addl-tgt-count  /* count of addl items added to tgt-list */
   int: build-count /* count of number of segments built for ISRT op */
   int: ssa-count /* count of the number of ssa's in multiple ISRT */

   char: cmd-code /* command codes 'D', 'F', or 'V' */
   char: attr-code  /* 's'=CHAR, 'i'=INT, 'f'=FLOAT, from schema */
   char: data-type
   str: label-name
   str: segment-name
   str: field-name
   str: abdl-str
   str: temp-str
   flag: loop-flag  /* there's a GOTO 'label' loop in curr request */
   list: tgt-list  /* list of sequence field attribute names */
   list: insert-list  /* list of attribute-value pairs for ISRT op */
   list: insert-nodes /* list Of insert-list(s) for multiple ISRT op */

%}
```

```
%token   /* List All Tokens From "LEX", and their TYPE, here */

%start statement

%%

statement: dml-statement
        {
        if (! spec-ret-op)
          save last curr-seg-ptr
        end-if
        return
        }
       ddl-statement
        {
        return
        }
        ;

ddl-statement: db-desc segment-list
              ;
db-desc: DBD
        {
        creating = TRUE
        curr-1st-child-ptr = NULL
        }
        NAME EQ db-name
        {
        locate dbid schema header node
        if (db names do not match)
          print ("Error - given db-name doesn't match db-name in file")
          perform yyerror()
          return
        endif
        }
        ;

segment-list: segment-desc
          segment-list segment desc
           ;

segment-desc: segment field-list
           ;
```

```
segment: SEGM NAME EQ
        {
        allocate and init a new segment-node
        dbid-node num-seg++
        }
        segment-spec
        ;

segment-spec: segment name
        {
        if (! valid-child (segment-name, curr-1st-child-ptr))
           copy segment-name to current segment-node
        end-if
        else
           print ("Error - 'segment-name' segment doubly defined in db")
           perform yyerror()
           return
        end-else
        }
       A
        ;
```

```
A: empty
  {
  connect new segment-node to the dbid root-ptr
  }
  COMMA PARENT EQ segment-name
  seg-ptr = the root segment of the db
  if (valid-parent(seg-ptr,segment-name,parent-ptr))
    connect the new segment-node to the appropriate parent-node
    establish curr-1st-child-ptr
    parent-node num-child++
    first-child and sibling node(s) num-sib++
  end-if
  else
    print ("Error - 'segment-name' parent-node does not exist")
    perform yyerror()
    return
  end-else
  }
  ;

field-list: field-desc
         {
         connect new attr-node to segment-node
         }
         field-list field-desc
         {
         connect successive attr-node(s) to segment-node
         }
         ;

field-desc: FIELD NAME EQ
         {
         allocate and init a new attr-node
         segment-node num-attr++
         }
         field-spec
         ;
```

```
field-spec: field-name
    {
    if (valid-attribute(seg-ptr,field-name,&attr-len,&attr-type))
     print("Error - 'field-name'attr doubly defined in 'segment-name'")
     perform yyerror()
     return
    endif
    else
    copy field-name to attr-node
    end-else
    attr-node key-flag = 0
    attr-node multiple field = 0
    }
   COMMA field-data
   LPAR field-name
    {
    if (valid-attribute(seg-ptr,field-name,&attr-len,attr-type))
     print("Error - 'field-name'attr doubly defined in 'segment-name'")
     perform yyerror()
     return
    end-if
    else
    copy field-name to attr-node
    end-else
    }
   COMMA SEQ B RPAR COMMA field-data
    {
    attr-node key-flag = 1
    }
    ;

B: empty
    {
    attr-node multiple-field = 0
    }
    COMMA M
    {
    attr-node multiple-field = 1
    }
    ;

field-data: C BYTES EQ INTEGER
        {
        attr-node length = INTEGER
        }
        ;
```

```
C: empty
  {
  attr-node type = 's'   /* default condition */
  }
  TYPE EQ data-type COMMA
  ;

data-type: CHAR
      {
      attr-node = 's'
      }
      INT
      {
      attr-node type = 'i'
      }
      FLT
      {
      attr-node= 'f'
      }
      ;

dml-statement: J ssa
      {
      if ( (label) and (! goto-found) )
       print ("Warning - 'label-name' label defined, but not referenced")
       perform yyerror()
       return
      end-if
      if (operator-flag = ISRT)
       if ( (single-build) and (star-d) )
        print ("Error - '*D' cmd code implies a multiple seg ISRT")
        perform yyerror()
        return
      end-if
      if ( (! single-build) and (! star-d) )
       print ("Error - '*D' cmd code req'd for multiple seg ISRT")
       perform yyerror()
       return
      end-if
      if (! single-build)
       if (build-count ! = ssa-count)
        print ("Error - num of segs built not equal to num ssa's")
```

```
   perform yyerror()
   return
  end-if
  for(each node in the list of insert-lists)
   if (! single-build)
    re-establish the saved curr-seg-ptr and segment-name
   end-if
   if (insert-attrs < 1)
    copy all segment attrs to insert-list and count insert-attrs
   end-if
   if (insert-attr ! = insert-vals)
    print ("Error - too many, or not enough values inserted")
    perform yyerror()
    return
   end-if
   for (each attribute in the curr-seg)
    if (segment attribute missing from insert list)
     add item to insert-list with default values
     of 'Zz' if type CHAR, and '0' if type INT
    end-if
    else
     if (insert-list item = seq-fld of curr-seg)
      seq-fld-has-value = TRUE
     if (! seq-fld-has-value)
      print ("Error - seq-fld value req'd in ISRT op")
      perform yyerror()
      return
     end-if
     if (! valid-attribute (curr-seg-ptr, field-name, &attr-len,
         &attr-type))
      print("Error-'field-name' attr does not exist in 'segment-name
            segment")
      perform yyerror()
      return
      end-if
      if (insert-list data-type ! = attr-type)
       print ("Error - 'field-name' attr must be type 'attr-type'")
       perform yyerror()
       return
      end-if
      if (attr-len < strlen(insert-list value))
       print ("Error - 'field-name' attr max length = 'attr-len'")
       perform yyerror()
       return
      end-if
  end-else
end-for
```

```
    if ( (seq-fld-has-value) and (single-build) )
     delete last abdl-str present
     (ie, seq-fld given, no retrieve request required)
    end-if
    alloc and init a new abdl-str
    copy "°INSERT (<TEMPLATE,'segment-name'>" to abdl-str
    if (single-build)
     for (each item in tgt-list)
      concat ",<'seq-fld',***...***>" to abdl-str
     end-for
    end-if
    else
     for (each node in the list of insert-lists)
      concat ",<'first attr-name', 'attr-value'> " to abdl-str
     end-for
    end-else
    for (each item in insert-list)
     concat ",<'attr-name', 'attr-value'> " to abdl-str
    end-for
    concat ") " to abdl-str
  end-for
end-if

if ( (! spec-ret-op) and (single-build) )
  for (all abdl-str(s), except the last one)
    concat tgt-list and BY-clause to RETRIEVE reqs
    (ie, "('tgt-list') BY 'seq-fld' ")
    if (operator-flag = ISRT) and (cmd-code = StarD)
     retrieve all attr names and add to tgt-list
    end-if
  end-for

  concat all attrs to last RETRIEVE request
  concat ") BY 'sequence-field'  to last RETRIEVE request

  if (operator-flag = DLET)
   form the descendant deletes to complete the DLET req
  end-if
end-if

if (spec-ret-op)
  if (operator-flag = GN)
    form the descendant retrieves to complete the GN (no ssa) req
  end-if
  else if (operator-flag = GNP)
  form retrieves for children to complete the GNP (no ssa) req
  end-else-if
  else
   print ("Error - seg-name must be specified if GN or GNP op")
   perform yyerror()
   return
  end-else
 end-if
 }
 ;

J: empty
  H
   {
   label = TRUE
   save label-name ('H') for later comparison with GOTO statement
   }
   ;

dli-operator: empty  GU GN GNP GHU GHN GHNP build-segs ISRT
   {
   set appropriate operator-flag
   }
   ;

build-segs: build-segment
   {
   build-count = build-count + 1
   }
   build-segs build-segment
   {
   build-count = build-count + 1
   single-build = FALSE
   }
   ;

build-segment: BUILD I COLON
   {
   inserting = TRUE
   }
   value-list
   ;

I: empty
   {
   alloc and init insert-list node
   }
   LPAR field-name-list RPAR
   ;
```

```
field-name-list: field-name
        {
        alloc and init insert-list node
        alloc first insert-list item
        copy 'field-name' to insert-list
        insert-attrs++
        }
       field-name-list COMMA field-name
        {
        alloc next insert-list item
        copy 'field-name' to insert list
        insert-attrs++
        }
        ;

value-list: LPAR constant-list RPAR
        {
        inserting = FALSE
        if (insert-attrs > 0)
         if (insert-attrs ! = insert-vals)
          print ("Error - too many, or not enough values inserted")
          perform yyerror()
          return
         end-if
        end-if
        }
        ;

constant-list: constant
        {
        if (insert-attrs < 1)
         alloc first insert-list item
        end-if
        if (literal-const)
         convert-AlphaNumFirst ('constant')
         literal-const = FALSE
         copy data-type = 's' to insert-list
        end-if
        else
         copy data-type = 'i' to insert-list
        end-else
        copy 'constant' to insert-list
        insert-vals++
        }
```

```
constant-list COMMA constant
  {
  if (insert-attrs < 1)
   alloc next insert-list item
  end-if
  if (literal-const)
   convert-AlphaNumFirst ('constant')
   literal-const = FALSE
   copy data-type = 's' to insert-list
  end-if
  else
   copy data-type = 'i' to insert-list
  end-else
  copy 'constant' to insert-list
  insert-vals++
  }
  ;

E: empty
  {
  if (label)
   print ("Warning - 'label-name' label defined, but not referenced")
  end-if
  }
 GOTO H
  {
  goto-found = TRUE
  if ( (! label) or ((label) and ('H'! = 'label-name')))
   print ("Error - label for "GOTO H' not defined"))
   perform yyerror()
   return
  end-if
  if (op-flag ! = GnOp, or GnpOp, or IsrtOp)
   print ("Error - loops used only w/GN,GNP, or ISRT operations")
   perform yyerror()
   return
  end-if
  if ( (! spec-ret-op) and (single-build) )
   set loop-flag for use by KC
  end-if
  else if (! single-build)
   print ("Error - loops cannot be used w/ multiple ISRT ops")
   perform yyerror()
   return
  end-else-if
  }
```

```
NFGOTO H
 {
 goto-found = TRUE
 if ( (! label) or ((label) and ('H' ! = 'label-name')))
  print ("Error - label for 'NFGOTO H' not defined")
  perform yyerror()
  return
 end-if
 if (op-flag ! = GnOp, or GnpOp, or IsrtOp)
  print ("Error - loops used only w/ GN, GNP, or ISRT operations")
  perform yyerror()
  return
 end-if
 if ((! spec-ret-op) and (single-build))
  set loop-flag for use by KC
 end-if
 else if (! single-build)
  print ("Error - loops cannot be used w/ multiple ISRT ops")
  perform yyerror()
  return
 end-else-if
 }
 ;

K: empty
  dli-op
   ;

dli-op: DLET
    {
    if (not preceded by a GET HOLDoperation
     print ("Error - DLET must be precededby GHU, GHN, or GHNP")
     perform yyerror()
     return
    end-if
    else
    op-flag = DLET
     alloc and init a new abdl-str
     /* formulate the first DELETE request */
     copy "°DELETE ((TEMPLATE = 'segment-name')" to abdl-str
     for (ea item in the tgt-list)
      concat " and ('item-name' = ***...***)" to abdl-str
     end-for
     concat ")  to abdl-str
    end-else
    }
```

```
chg-pred REPL
  {
  if (not preceded by a GET HOLD operation)
   print ("Error - REPL must be preceded by GHU, GHN, or GHNP")
   perform yyerror()
   return
  end-if
  else
   op-flag = REPL
  end-else
  }
  ;

chg-pred: CHANGE
  {
  updating = TRUE
  }
  field-name TO constant
  {
  updating = FALSE
  if (! valid-attribute(curr-seg-ptr,field-name,&attr-len,&attr-type))
   print ("Error- 'field-name' attr does not exist in 'segment-name'
          segment")
   perform yyerror()
   return
  end-if
  if (literal-const)
   convert-AlphaNumFirst('constant')
   literal-const = FALSE
   data-type = 's'
  end-if
  else
   data-type = 'i'
  end-else
  if (data-type ! = attr-type)
   print ("Error - 'field-name' attr must be type 'attr-type'")
   perform yyerror()
   return
  end-if
  if (attr-len < strlen('constant')
   print ("Error - 'field-name' attr max length = 'attr-len'")
   perform yyerror()
   return
  end-if
  alloc and init a new abdl-str
  copy "°UPDATE ((TEMPLATE = 'segment-name')" to abdl-str
  for (each item in tgt-list)
   concat " and ('seq-fld' = ***...***)" to abdl-str
  end-for
  concat ") <'field-name' = 'constant'> "to abdl-str
  }
  ;

ssa: seg-srch-arg
   {
   ssa-count = ssa-count + 1
   prev-seg-ptr = curr-seg-ptr
   if ((operator-flag = ISRT) and (! singl-build))
    save the curr-seg-ptr and segment-name in first insert-list node
   end-if
   first-ssa = FALSE
   }
   ssa seg-srch-arg
   {
   ssa-count = ssa-count + 1
   if (the parent of the curr-seg-ptr = prev-seg-ptr)
    prev-seg-ptr = curr-seg-ptr
   end-if
   else
    print ("Error - SSA specifies incorrect hierarchical path")
    perform yyerror()
    return
   end-else
   if ((operator-flag = ISRT) and (! single-build))
    save the curr-seg-ptr and segment-name in first insert-list node
   end-if
   }
   ;

seg-srch-arg: dli-operator segment-name
    {
    seg-ptr = the root of the db
    if (! valid-parent(seg-ptr,'segment-name',curr-seg-ptr))
     print ("Error - 'segment-name' segment does not exist")
     perform yyerror()
     return
    end-if
    if ((operator-flag ! = ISRT) or (single-build))
     alloc and init a new abdl-str and a new tgt-lidt item
     copy "°RETRIEVE (" to abdl-str
     copy segment sequence field and length to tgt-list
    end-if
```

```
        if ((label) and (not-marked))
         not-marked = FALSE
         label-ptr = current abdl-str
        end-if
        if ((first-ssa) or (missing-root))
         if (curr-seg-ptr = root of the db)
          insert seq-fld(s) to tgt-list for all parents/grandparents
          addl-tgt-count = number inserted
          missing root = TRUE
         end-if
        end-if
        save 'segment-name' for later use
        }
        L G
        {
        delete first 'addl-tgt-count' items from tgt-list
        addl-tgt-count = 0
        if (single-build)
         concat ") " to abdl-str
        end-if
       E K
       }
       dli-operator
       {
       spec-ret-op = TRUE;
       }
       E K
       ;

L:  empty
    ASTERISK N
    ;

N:  D F V
    {
    set cmd-code to appropriate code (StarF, or StarV)
    if (N is D)
     star-d = TRUE
     if (single-build)
      set cmd-code to StarD
     end-if
    end-if
    }
    ;
```

```
G: empty
  {
  if (! single-build)
   do nothing
  end-if
  else
   if (curr-seg-ptr = root of the db)
    concat "(TEMPLATE= 'segment-name'" to abdl-str
   end-if
   else
    concat "(TEMPLATE = 'segment-name')" to abdl-str
    for (ea item in tgt-list)
     concat " and ('item-name' = ***...***)" to abdl-str
    end-for
   end-else
  end-else
  }
  LPAR boolean RPAR
  {
  if (or-where)
   concat ")" to abdl-str
   or-where = FALSE
  end-if
  }
  ;

boolean: boolean-term
  {
  concat "(TEMPLATE = 'segment-name') and " to abdl-str
  form symbolic id predicates:"('seq-fld'=***...***)" from tgt-list,
   for all previous segments, and concat them to the abdl-str, each one
   separated by " and ".
  concat temp-str to abdl-str
  }
  boolean OR
  {
  or-where = TRUE
  abdl-str°11  = '('
  concat ") or ((TEMPLATE = 'segment-name') and " to abdl-str
  copy 'empty str' to temp-str
  }
  boolean-term
  {
  form symbolic id predicates: "('seq-fld'= ***...***)" from tgt-list,
   for all previous segments, and concat them to the abdl-str, each one
   separated by " and ".
  concat temp-str to abdl-str
  }
  ;
```

```
boolean-term: boolean-factor
   boolean-term AND
   {
   and-where = TRUE
   concat " and " to temp-str
   }
   boolean-factor
   ;

boolean-factor: predicate
   ;

predicate: field-name
  {
  if (! valid-attribute(curr-seg-ptr,field-name,&attr-len,&attr-type))
   print ("Error - 'field-name' attr does not exist in 'segment-name'
          segment")
   perform yyerror()
   return
  end-if
 else
   if ((! and-where) and (! or-where))
    alloc temp-str
    copy "(" to temp-str
   end-if
   else
    concat "(" to temp-str
   end-else
   concat 'field-name' to temp-str
   save 'field-name' for later use
   and-where = FALSE
  end-else
  }
  comparison
  {
  concat " 'comparison'" to temp-str
  }
  constant
  {
  if (literal-const)
   convert-AlphaNumFirst('constant')
   literal-const = FALSE
  end-if
  concat "'constant')" to temp-str
  }
  ;
```

```
comparison: EQ NE LT GT GE
  ;

constant: QUOTE H QUOTE
    {
    literal-const = TRUE
    if ((! inserting) and (! updating))
     if (attr-type ! = 's')
      print ("Error - 'field-name' attr must be type INT")
      perform yyerror()
      return
     end-if
     if (attr-len < strlen('H'))
      print ("Error - 'field-name' attr max length = 'attr-len'")
      perform yyerror()
      return
     end-if
  end-if
    }
INTEGER
    {
    if ((! inserting) and (! updating))
     if (attr-type ! = 'i')
      print ("Error - 'field-name' attr must be type CHAR")
      perform yyerror()
      return
     end-if
     if (attr-len < strlen('INTEGER'))
      print ("Error - 'field-name' attr max length = 'attr-len'")
      perform yyerror()
      return
     end-if
    end-if
    }
    ;

H: IDENTIFIER
   VALUE
    ;

db-name: IDENTIFIER
    ;

segment-name: IDENTIFIER
    ;

field-name: IDENTIFIER
    ;
```

```
empty: ;

end-proc yyparse


%%


proc yyerror(s)
    char s
  if (operation = CreateDB)
   set error flag for the LIL
   print ("Error - DBD Description file for 'curr-seg' in error")
   free all the malloc'd variables in the current schema
  end-if
  else
   set error flag for the LIL
   free all the malloc'd variables in the kms data structures
  end-else

  reset all boolean and counter variables
  print (s)
end-proc yyerror
```

```
proc MATCH()
  /* Th_s routine checks the operator flag for the incoming*/
  /* transaction and branches to the appropriate DL/I operation */

  kms-list: list;
  sit-list: list;
  status-list: list:
  head-kms-list-ptr: ptr;
  head-sit-list-ptr: ptr;
  status-ptr: ptr;
  sit-ptr: ptr;
  first-status-node-ptr: ptr;
  curr-status-node-ptr: ptr;

  /* the kms list cannot be null  */
  if (kms-list <> 'null')
   case (kms-list.operation)
     "GhuOp" : /* Get Hold Unique operator */
               perform GET-HOLD-UNIQUE();

     "IsrtOp" : /* Insert operator  */
               perform INSERT();

     "GuOp"  : /* Get Unique operator  */
               perform GET-UNIQUE();

     "GnOp"  : /* Get Next operator  */
               perform GET-NEXT();

     "SpecRetOp" : /* Special Retrieve operator  */
               perform SPECIAL-RETRIEVE();

     "GnpOp" : /* Get Next Within Parent operator  */
               perform GET-NEXT-PARENT();
   end-case;

 end-proc;
```

```
proc GET-HOLD-UNIQUE()
  /* A GHU operation allows one user exclusive access to the database */
  /* so that subsequent deletes or replaces will occur before any     */
  /* further users can access the database                            */

  dlet-flag: int; /* boolean flag to tell if found a DELETE op */
  done: int;      /* boolean flag  */

if (sit-list <> 'null')
 print ("Error - sit-list is not null as required for GhuOp");
else
 /* When a GHU is found, the type of operation must be identified.    */
 /* The kms list is scanned looking for a delete operator.  If found, */
 /* a status node is created and set to point to the first node of    */
 /* the kms list (the GHU).  A second status node is created and set  */
 /* to the kms node that has the beginning-of-request delete flag     */
 /* set.  If the delete operator is not found, this indicates a       */
 /* replace operation or a list with nothing butGHU's.  In this case */
 /* a status node is created and set to point to the first node of    */
 /* the kms list (the GHU).                                */

 done = 'false';
 dlet-flag = 'false';
 sit-ptr = kms-list + 1;

 /* walk down sit list until find DELETE operator or empty list */
 while (sit-ptr <> 'null' OR not done)
  if (sit-ptr.operation = DletOp)
   /* case of DELETE operation */
   if (first-status-list-ptr = 'null')
    /* case of status list being empty */
    allocate a new status node;
    first-status-list-ptr = new status node;
    curr-status-list-ptr = new status node;
    status-ptr = head-kms-list ptr;
    allocate a new status node;
    append status node to the status list;
    status-ptr = sit-ptr;
   end-if;
   else
    print ("Error - status list not null as required for GhuOp");
   end-else;
   dlet-flag = 'true';
   done = 'true';
  end-if
  sit-ptr = sit-ptr + 1;
 end-while;
```

```
     if (dlet-flag = 'false')
      /* case that no DELETE operators were found in sit list; this  */
      /* indicates that we have REPLACE operations or just GHU's     */
      allocate a new status node;
      first-status-node-ptr = new status node;
      curr-status-node-ptr = new status node;
      status-ptr = head-kms-list-ptr;
     end-if;

     /* set sit list ptr to heading of the kms list and null out ptr */
     head-sit-list-ptr = head-kms-list;
     head-kms-list = 'null';
    end-else;

  end-proc;

  proc INSERT()
    /* An insert operation is used to add a new segment, "node" */
    /* to the database.                  */

    first-bor: int; /* boolean flag set when beginning-of-req found  */

    /* An insert operation can only access the database from the root. */
    /* As long as the sit list is null, then we know that the currency */
    /* pointer is pointing to the root.                     */
    if (sit-list <> 'null')
     print ("Error - sit list is not null as required for IsrtOp");
    else
     /* set sit ptr to the head of the kms list  */
     sit-ptr = head-kms-list-ptr;
     first-bor = TRUE;
```

```
/* walk down the kms list until it is empty */
 while (sit-ptr <> 'null')
 /* When an insert is detected, the kms list is scanned and a  */
 /* status node is created and set to point to each kms node   */
 /* that contains a beginning-of-request flag.  The kms list   */
 /* is then transferred to the sit list before exiting.        */

 if (sit-ptr.BOR = 'true')
  allocate a new status node;
  if (first-bor = 'true')
    /* case of the status node being the first on the list */
   first-status-node-ptr = new status node;
   curr-status-node-ptr = new status node;
   first-bor = 'false';
  end-if;
  else
   append the status node onto status list;
  end-else;

  /* fill in the status node's contents */
  status-ptr = sit-ptr;
 end-if;

 sit-ptr =sit-ptr + 1;
end-while;

 /* set sit list ptr to head of the kms list and null out kms ptr */
 head-sit-list-ptr = head-kms-list-ptr;
 head-kms-list-ptr = 'null';
end-else;

end-proc;
```

```
proc GET-UNIQUE()
  /* A GU operation is used to access the database from the  */
  /* root of the database.                          */

 /* A GU operation can only access the database from the root.  */
 /* As long as the sit list is null, then we know that the      */
 /* currency pointer is pointing to the root.               */
 if (sit-list <> 'null')
  print ("Error - sit-list not null as required for GuOp");
 else
  if (first-status-node-ptr= 'null')
   /* When a legitimate GU is found, we are sure that the   */
   /* currency of the request is correct.  In this case we  */
   /* simply transfer the sit list for the GU from the kms  */
   /* list to the sit list and create a single status node  */
   /* that points to the first request of the GU.         */

   /* allocate a status node */
   allocate a new status node;

   /* set head of the status list to the allocated node */
   first-status-node-ptr = new status node;
   curr-status-node-ptr = new status node;

   /* fill in the contents of the allocated node */
   status-ptr = head-kms-list-ptr;

   /* set sit list ptr to heading of the kms list and null out kms ptr */
   head-sit-list-ptr = head-kms-list-ptr;
   head-kms-list-ptr = 'null';
  end-if;
  else
   print ("Error - status list not null as required for GuOp");
  end-else;

 end-else;
sp.
end-proc;
```

```
proc GET-NEXT()
  /* A GN operation is used to access the next lower level of the    */
  /* database.  It is used only after a GU operation has established */
  /* a currency pointer to a specific level of the database.         */

  found, done;     /* boolean flags */

  prev-kms-ptr;     /* ptr to the previous node on kms list */
  prev-sit-ptr;     /* ptr to the previous node on the sit list */

if (head-sit-list-ptr = 'null')
 /* With the sit list being null, a GN is the same as a GU if the name */
 /* of the first node of the kms list is the same as the root segment. */
 if (head-kms-list-ptr.seg-name = root segment name        */
  perform GET-UNIQUE();
 end-if;
 else
  print ("Error - currency pointer must be set to the root");
  print (" or specify complete path");
 end-else;

else
 if (first-status-node-ptr = 'null')
  print ("Error - status list null for GnOp");
 end-if;
 else
  /* When a valid GN is found, we know the segment that we want is     */
  /* the next occurrence of a legitimate child or the segment the      */
  /* currency pointer points to.  If the segment is a child then we    */
  /* create a status node pointing to the first node of the kms list.  */
  /* By being a child we guarantee ourselves that part of the kms      */
  /* list matches some of the sit list so the status field of the      */
  /* allocated status node is set to MATCHPART. If the segment is not  */
  /* a child but the current segment, the amount of overlap between    */
  /* the sit list and the kms list must be checked. The parent         */
  /* pointer of the first kms node is set to the node above the node   */
  /* that it matches in the sit list. A status node is created         */
  /* pointing to the first node of the kms list. The kms and sit       */
  /* lists are then checked to see how much overlapthey contain. The   */
  /* status field is set to MATCHPART or MATCHALL as appropriate.      */

 dli-ptr->di-saved-seg-ptr-2->hn-first-child->hn-name);
 if (head-kms-list-ptr.seg-name is a valid child of the node
            currently pointed to by the currency pointer)
  /* segment we want the next of is a child of the current segment  */
  status-ptr = head-status-node-ptr;
```

```
/* walk down to the end of the status list*/
 while (status-ptr.next <> 'null')
  status-ptr = status-ptr + 1;
 allocate a new status node;
 status-ptr = head-kms-list;
 status-ptr.status = MATCHPART;
 append status node to the status list;

 /* walk down to the end of the sit list */
 sit-ptr = head-sit-list-ptr;
 while (sit-ptr.next <> 'null')
  sit-ptr = sit-ptr + 1;

 /* append the kms list to the end of the sit list  */
 sit-ptr.next = head-kms-list-ptr;
 head-kms-list-ptr = 'null';
end-if;
else
 /* check to see where first node in kms list overlaps sit list, */
 /* i.e., if the node is a descendent of the current node       */
 found = 'false';

 /* set pointer to the head of the sit list */
 sit-ptr = head-sit-list-ptr;
 while (sit-ptr <> 'null' AND not found)
  if (sit-ptr.seg-name = head-kms-list-ptr.seg-name)
   found = TRUE;
  end-if;
  else
   sit-ptr = sit-ptr + 1;
 end-while;

 if (! found)
  print ("Error - match not found in GnOp");
 end-if;
 else
  /* found a valid overlap so set the parent pointer of the  */
  /* first node of the kms list to the node above the node   */
  /* in the sit list that matched.                           */
  head-kms-list-ptr.parent = sit-ptr.prev;

  /* walk down to the end of the status list */
  status-ptr = first-status-node-ptr;
  while (status-ptr.next <> 'null')
   status-ptr = status-ptr + 1;
```

```
allocate a new status node;
append the status node to the status list;
kms-ptr = head-kms-list-ptr;
prev-kms-ptr = kms-ptr;
prev-sit-ptr = sit-ptr;
done = 'false';
while (not done)
 /* Nowwe walk down the kms and sit lists to see how  */
 /* much overlap the sit list contains                */
 free (kms-ptr.result-file;
 kms-ptr.result-file = sit-ptr.result-file;
 kms-ptr = kms-ptr + 1;
 sit-ptr = sit-ptr + 1;
 if (kms-ptr = 'null' OR sit-ptr = 'null')
   done = 'true';
 end-if;
 else
  /* Both lists still contain nodes so increment them */
  prev-sit-ptr = sit-ptr;
  prev-kms-ptr = kms-ptr;
 end-else;
end-while;

if (kms-ptr = 'null')
 /* case where sit list contained all of kms list  */
 status-ptr = prev-kms-ptr;
 status-ptr.status = MATCHALL;
end-if;
else
 /* case where sit list contained part of the kms list */
 status-ptr = kms-ptr;
 status-ptr.status = MATCHPART;
end-else;

/* Now append kms list to the end of the sit list */
if (sit-ptr = 'null')
 prev-sit-ptr.next = head-kms-list-ptr;
end-if;
else
 while (sit-ptr.next <> 'null')
  sit-ptr = sit-ptr + 1;
 sit-ptr.next = head-kms-list-ptr;
end-else;
```

```
          head-kms-list-ptr = 'null';
       end-else;

     end-else;

    end-else;

  end-else;

end-proc;

proc SPECIAL-RETRIEVE()
   /* When a GN or GNP operation has been selected without any segment */
   /* search arguments specified, the normal GN or GNPoperation of    */
   /* returning the next segment occurrence is skipped.  Instead we    */
   /* consider this a special retrieve to return all segment           */
   /* occurrences below the segment the currency pointer points to.    */

  if (head-sit-list-ptr = 'null')
   print ("Error - status list null for SpecRetOp");
  end-if;
  else
   /* walk down to the end of the status list  */
  status-ptr = first-status-node-ptr;
  while (status-ptr.next <> 'null')
   status-ptr = status-ptr + 1;

  allocate a new status node;
  status-ptr = head-kms-list-ptr;
  append status node to the status list;

  /* walk down to the end of the sit list - the last node */
  /* represents the current segment     */
  sit-ptr = head-sit-list-ptr;
  while (sit-ptr.next <> 'null')
   sit-ptr = sit-ptr + 1;

  /* any nodes in the kms list with parent pointer = null    */
  /* must have the parent pointer set to the current segment */
  kms-ptr = head-kms-list-ptr;
  while (kms-ptr <> 'null')
   if (kms-ptr.parent = 'null')
    kms-ptr.parent = sit-ptr;
   end-if;
   kms-ptr = kms-ptr + 1;
 end-while;
```

```
  /* append the kms list to the end of the sit list */
  sit-ptr.next = head-kms-list-ptr;
  head-kms-list-ptr = 'null';
 end-else;

end-proc;

proc GET-NEXT-PARENT()
 /* A GNP operation is used to access the database just below the  */
 /* current node the currency pointer is pointing to, rather than  */
 /* having to specify the access path from the root as in a GU.    */

if (head-kms-list-ptr.cmd-code <> 'StarF')
 perform GET-NEXT();
end-if;
else
 /* Once a valid GNP operation has been detected, we are sure the  */
 /* currency pointer is set to a segment (node) somewhere in the   */
 /* hierarchy with legitimate children beneath it. We then take    */
 /* the parent pointer of the first node of the kms list and set   */
 /* it to the next to last node of the sit list.  A status node is */
 /* then created and set to point to the first node of the kms     */
 /* list. Finally, the kms list is appended to the sit list.       */

 /* walk down to the end of the sit list */
 sit-ptr = head-sit-list-ptr;
 while (sit-ptr.next <> 'null')
  sit-ptr = sit-ptr + 1;

 /* Check to see if the head of the kms list is a valid child of the */
 /* next to last node of the sit list            */
 if (head-kms-list-ptr.seg-name is not a valid child of the next
                       to last node in the sit list)
  print ("Error - valid child not found");
 end-if;
 else
  /* Since it is a valid child, set the parent pointer of the first  */
  /* node of the kms list to the next to last node of the sit list   */
  head-kms-list-ptr.parent = sit-ptr.prev;
```

```
    /* walk down to the end of the status list */
    status-ptr = first-status-node-ptr;
    while (status-ptr.next <> 'null')
     status-ptr = status-ptr + 1;

    allocate a new status node;
    status-ptr = head-kms-list-ptr;
    append the status node to the status list;

    /* Appendthe kms list to the end of the sit list */
    sit-ptr.next = head-kms-list-ptr;
    head-kms-list-ptr = 'null';
   end-else;

  end-else;

  end-proc;
```

LIST OF REFERENCES

[1] Demurjian, S.A. and Hsiao, D.K., "New Directions in Database Systems Research and Development," in the *Proceedings of the New Directions in Computing Conference*, Trondheim, Norway, 1985; also in Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

[2] Hsiao, D.K. and Haray,F., "A Formal System for Information Retrieval from Files," *Communications of the ACM*, Vol.13, No. 2, February 1970.

[3] Banerjee, J. and Hsiao, D.K., "DBC Software Requirements for Supporting Relational Databases," Technical Report, OSU-CISRC-TR-77-7, The Ohio State University, Columbus, Ohio, November 1977.

[4] Rollins, R., "Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

[5] Banerjee, J. and Hsiao, D.K., "The Use of a Database Machine for Supporting Relational Databases," *Proceedings 5th Workshop on Computer Architecture for Nonnumeric Processing* (August 1978).

[6] Banerjee, J., Hsiao, D.K., and Ng, F., "Database Transformation, Query Translation and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, (January 1980).

[7] Banerjee, J. and Hsiao, D.K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," *Proceedings of National ACM Conference* (1978).

[8] Macy, G., "Design and Analysis of an SQL Interface for a Multi-Backend Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1984.

[9] Kloepping, G.R. and Mack, J.F., "The Design and Implementation of a Relational Interface for the Multi-Lingual Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[10] Weisher, D., "Design and Analysis of a Complete Hierarchical Interface for a Multi-Backend Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

[11] Benson, T.P. and Wentz, G.L., "The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[12] Wortherly, C.R., "The Design and Analysis of a Network Interface for a Multi-Backend Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[13] Emdi, B., "The Implementation of a CODASYL-DML Interface for a Multi-Lingual Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[14] IBM, *Information Management System IMS/360, Application Description Manual* (Version 2) (1971).

[15] Date, C.J., in *An Introduction to Database Systems,* (Addison-Wesley, 1981), 3rd edition.

[16] Coker, H., Demurjian, S.A., Hsiao, D.K., Rodeck, B.D., and Zawis, J.A., "The Multi-Model Database System," Technical Report, Naval Postgraduate School, Monterey, California, July 1987.

[17] Rodeck, B., "Accessing and Updating Functional Databases Using CODASYL-DML," M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1986.

[18] Zawis, J.A., "Accessing Hierarchical Databases via SQL Transactions in a Multi-Model Database System," M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1987.

[19] Lesk, M.E. and Schimdt, E.,*Lex-A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill, New Jersey, July 1978.

[20] Coker, H., "Accessing A Functional Database Via CODASYL-DML Statements," M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
7.	Marciano Pitargue Vitalink C [illegible] unication Corporation 6607 Kais [illegible] e Fremont, CA [illegible]55	1
8.	William Sheehan 210 Maple Street Glens Falls, NY 12801	2